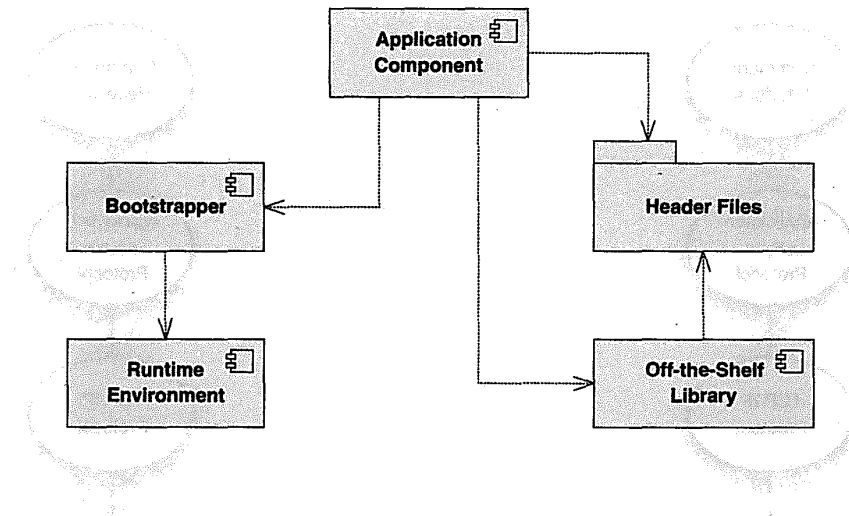


Figure 16-10.
UML diagram
without profile.



UML—The Unified Modeling Language

We have covered various aspects of UML (Booch, Rumbaugh, and Jacobson 2005) throughout earlier chapters of this book: its notation, its visualizations, its metamodel (Object Management Group 2002), and so on. This section will focus on UML as a standard and describe its advantages and disadvantages from that perspective.

UML provides a standard *syntax* for describing a wide variety of aspects of software systems. Basic UML (without extensions) prescribes very few semantics to go along with this syntax. It is possible for the same element or diagram to mean entirely different things. For example, the dashed arrow connector in UML indicates a dependency between two linked elements. That dependency could mean “needed to build” or “plugs-into” or “calls.” There is no way, without additional context or information, for a stakeholder to look at a dependency arrow in UML and have any clear idea what it means. This situation is depicted in Figure 16-10—the dependencies are all specified but the reader knows little about them.

UML that follows only the basic standard establishes a common symbolic representation of elements and their relationships. The fact that a system is described in UML implies little about the quality of that system—good designs can be documented in UML as easily as bad ones.

UML’s value can be substantially increased through the use of UML extensions and profiles. Recall that a UML profile is a coordinated set of extensions to UML. In a way, UML profiles are themselves standards that specialize the base UML standard for different domains. OMG, the organization that manages the UML standard, also manages a number of de jure standard UML profiles, including profiles for CORBA systems, enterprise application integration, and system engineering. The latter, called SysML, is discussed below.

Profiles may include stereotypes, tagged values, and constraints. UML stereotypes allow users to specialize existing UML symbols to add project- or domain-specific semantics. Stereotypes can specialize the dependency arrow to mean <<calls>> or

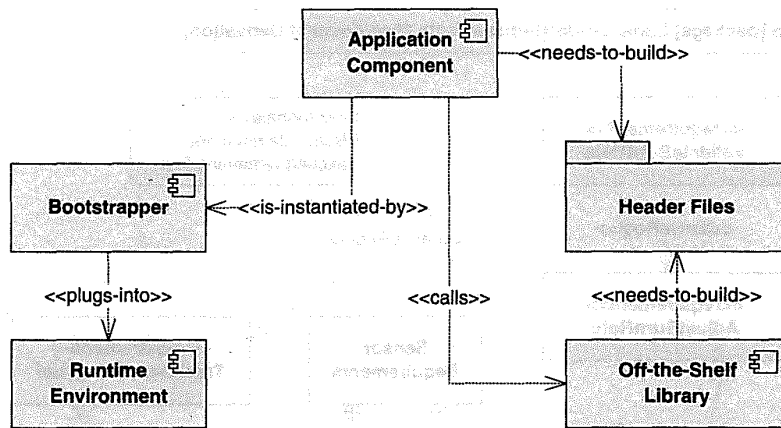


Figure 16-11.
UML diagram
with profile.

`<<needed-to-build>>` or `<<plugs-into>>`. For example, Figure 16-11 shows the same UML diagram as Figure 16-10, except the dependencies are stereotyped. This diagram is much more readable and provides substantially more information. To be fully useful, however, a UML profile for a project *must* be accompanied by detailed documentation explaining when and how to use the various stereotypes, tagged values, and constraints, as well as their meaning. With the proper use of profiles, a UML description becomes a valuable tool for stakeholder communication and system understanding, and also has positive impacts on traceability and analyzability.

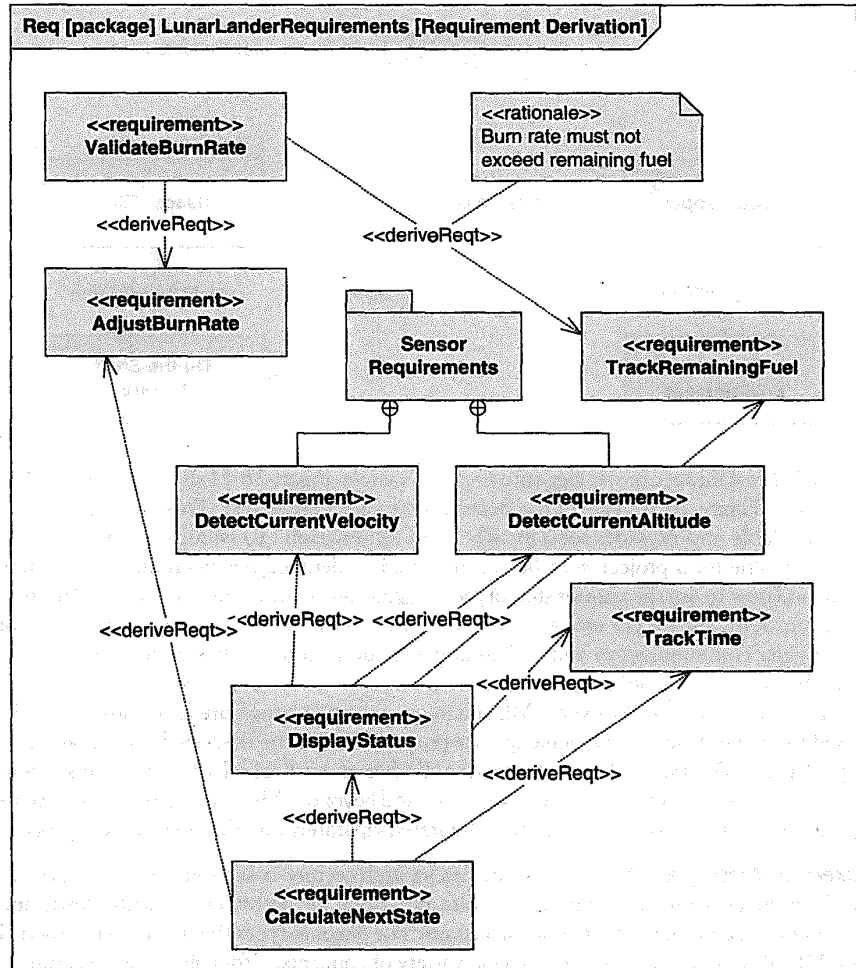
Even with profiles, however, UML and its accompanying tools are (currently) not well-suited to the verification of specific quality properties. Will the described system satisfy its requirements? Be efficient? Be cost-effective? Be free of deadlock? To determine answers to questions like these, additional methods above and beyond UML, ranging from inspections to testing to formal verification, must be integrated separately into the development process.

Takeaway Message. UML is not a panacea for architecture-based software development. UML provides a set of common, symbolic notations for architects to write down and communicate certain kinds of design decisions. The diagram types that have been provided in UML allow architects to model a wide variety of concepts, although there are certainly architectural design decisions that cannot be easily captured in UML. UML profiles should be used to improve the rigor and understandability of UML diagrams, but profiles have their limits. Profiles can decorate existing diagrams, but they cannot create entirely new diagrams. Modeling a system in UML makes few guarantees about that system—UML can be used to document both good and bad architectures (although UML does make it easier to understand, communicate about, and evaluate architectures).

16.2.3 SysML

Despite the wide variety of modeling concepts supported in UML, different engineering efforts require even more modeling breadth. A good example of this is SysML (SysML Partners 2005). SysML is an effort by a coalition of more than twenty engineering organizations (primarily corporations, but also universities and other organizations) (SysML.org 2007)

Figure 16-12.
SysML
requirement
diagram for
Lunar Lander.



to customize UML for systems engineering. With respect to standardization, the SysML community has taken an interesting two-pronged approach. One group is developing and promulgating an open-source version of the SysML standard under a license that allows users to redistribute and modify SysML on their own. From this open-source version, an official OMG standard version of SysML has been created. "OMG SysML" is trademarked, and evolves, like UML, through the standards committees and processes of the OMG.

Many individuals and organizations perceive UML to be too software-biased for modeling systems engineering concerns that include elements such as hardware, networks, development organizations, and so on. Of course, many UML diagrams can be interpreted in ways that do capture these concerns (for example, a traditional systems engineering block diagram closely resembles a class diagram in many ways). SysML includes a subset

of UML, but also extends it. Diagrams reused from UML are the activity, class, composite structure, package, sequence, state machine, and use case diagrams. Two of these diagrams are renamed for SysML: Class diagrams are called “block definition” diagrams and composite structure diagrams are called “internal block” diagrams. SysML also adds stereotypes to UML as well as two new diagram types: requirement and parametric diagrams. Requirement diagrams show system requirements and their relationships with other elements. Parametric diagrams show parametric constraints between structural elements, which are useful for performance and quantitative analysis.

Figure 16-12 shows a SysML requirement diagram for the Lunar Lander application. This diagram shows only requirements and their relationships; a more complex requirement diagram might actually include additional detail for each requirement such as the actual text describing the requirement, a unique ID, and so on. Requirement diagrams can also be used to associate requirements with elements from other diagrams such as use cases. Syntactically, requirement diagrams primarily reuse and specialize elements from class diagrams, using stereotypes on classes and relationships to make those elements specific to requirement diagrams.

Figure 16-13 shows a SysML parametric diagram. Parametric diagrams are specializations of SysML internal block diagrams—composite structure diagrams in UML. These diagrams allow architects to associate quantitative parameters with various model elements. These parameters can then be associated with mathematical models, equations, or analysis techniques that can be used to evaluate or assess parts of the system quantitatively. For example, several state-storage elements (the *Clock*, the *Lander State*) provide values to the next-state equation, which is used to calculate the next state of the Lunar Lander simulation.

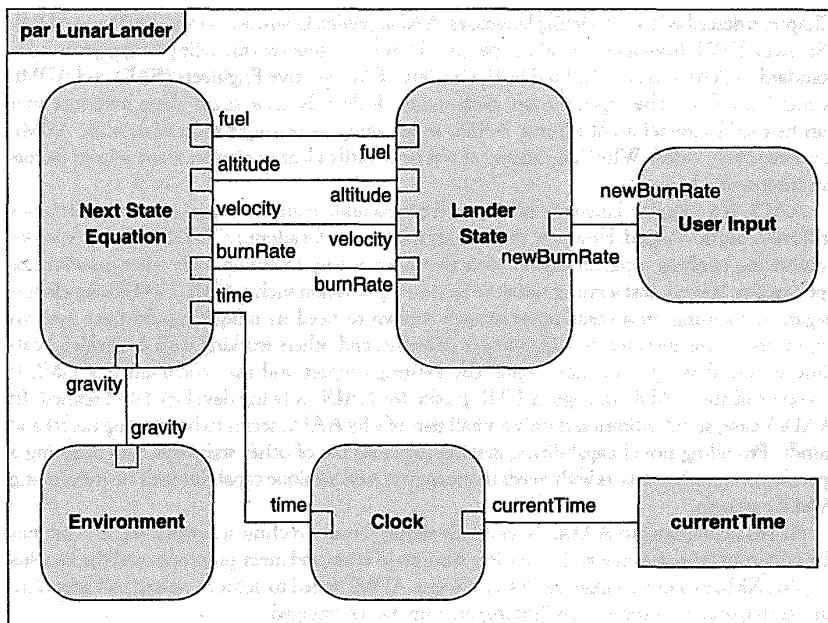


Figure 16-13.
SysML
parametric
diagram for
Lunar Lander.

The development of SysML is consistent with our earlier assertion that UML is not necessarily a complete solution for architectural modeling. The SysML developers left out certain UML diagrams they believed were extraneous or unnecessary, and added new diagrams that were missing. SysML has not been as universally supported or adopted as UML, despite the fact that it is not a radical departure from traditional UML 2. Tool support in the form of modeling environments, for example, has lagged behind UML, and while some UML modeling environments such as Telelogic Rhapsody have some explicit SysML support, many others do not. This indicates that even a large group of influential organizations cannot necessarily force the adoption of any particular standard. Note also that even the new SysML diagrams reuse and overload existing UML symbols, likely a compromise to provide some degree of modeling support in tools that are not SysML-aware.

Takeaway Message. SysML is an example of how standards can evolve, change, and fork apart. By reusing substantial parts of the UML standard, the SysML partners were able to save a tremendous amount of effort as well as allow users to apply their existing UML knowledge in a systems-engineering context. However, the changes and extensions they made to UML—conservative as they are—have been a double-edged sword. On one hand, they provide increased expressiveness for systems engineers. On the other hand, they have reduced the amount of tool support that exists for users. Even standards that are supported by a substantial amount of influential organizations must still win many hearts and minds on technical and business merits to achieve widespread success.

A Tale of Two Standards

Chapter 6 described two modeling languages, AADL (Feiler, Lewis, and Vestal 2003) and ADML (Spencer 2000), from a technical perspective. However, these two modeling languages are also standards—AADL is standardized by the Society of Automotive Engineers (SAE) and ADML is standardized by The Open Group. Both standards describe new, stand-alone notations that can be used to model architectures. AADL is becoming increasingly successful, while ADML has effectively failed. Why? The answer is not necessarily clear or simple; many related factors are interacting here.

AADL is a complex language. It has a rich syntax and, to an extent, graphical visualizations still are being developed. However, the primary motivation for adopting AADL is that it provides compelling analysis capabilities and tools that are not easy to obtain with other notations or approaches. It is not just a minor tweak to an existing notation such as UML. AADL's developers began by focusing on a community of users known to need its unique capabilities—systems engineers in the automotive and avionics domains, and others working with embedded, real-time system development. Leveraging the existing support and user community of UML is also part of the AADL strategy; a UML profile for AADL is being developed and refined. In AADL's case, standardization is only a small part of why AADL seems to be winning hearts and minds. Providing novel capabilities, making judicious use of other standards, and targeting a specific community of users with needs that match AADL's unique capabilities all are influencing AADL's growth.

ADML, compared to AADL, is relatively simple. From a technical perspective, it combines the core concepts of Acme with a few innovations in types and meta-properties and the benefits of using XML as a meta-language. As a standard, ADML failed to achieve widespread adoption, and anticipated tool support and interoperability never emerged.

It is difficult to say why. Unlike AADL, ADML's improvements over previous technologies are incremental. ADML was not targeted at a specific domain or user community, although some ADML documentation notes its usefulness in enterprise architecture modeling and points out deficiencies in UML in this regard. ADML may have also suffered because of the availability of contemporary alternatives such as Acme, Darwin, and xADL 1.0, lack of initial tools to help motivate community growth, and so on.

On a positive note, however, many of the concepts of ADML did eventually appear in UML 2 (and thus also in XMI). UML already had a huge user base, and these contributions shored up support for structural modeling that was considered to be a deficiency of UML 1.x. The ideas of the ADML standard were much better received when they became part of UML. Of course, defining a new standard like ADML is much easier than convincing the already-mature UML community to incorporate these new ideas. However, ADML's contributions were a natural fit, addressing well-known drawbacks of UML. Incorporating a complex standard with many new and conflicting ideas like AADL into UML would likely cause much more friction in the UML community, and may be infeasible.

There are several lessons to be learned from the AADL and ADML experience. Standardization alone does not ensure the success of an approach. Success is largely dependent on the size and passion of the user community surrounding the approach, and standardization is just one way to catalyze the development of a community. Another important element is to "prime the pump" with support for the standard: Developing initial tools, documentation, and training for a standard, even if they are not perfect, can entice early adopters much more than a simple specification. For incremental improvements, or improving deficiencies in an existing standard, it is often best to focus efforts on improving or evolving the existing standard rather than trying to develop a new, parallel standard.

16.2.4 Standard Tools

Many organizational cultures will standardize on a particular tool as much as (or more than) more abstract approaches. Corporations often evaluate a spectrum of tools and then purchase one (or a small number) of them in bulk, and that tool becomes mandatory for use within the company. Standard tools tend to be de facto standards, because they generally are developed and controlled by a single organization.

Standard UML Tools

The wide adoption of the UML standard generally induces organizations to adopt one or more standard tools for working with UML. This is often motivated by a desire for consistency and interoperability across an organization, as well as market pressures: Organizations buying tools from a single vendor can often get volume discounts and negotiate enhanced support agreements. UML itself, however, is not bound to any particular tool. Tools that have become de facto standards for UML editing include IBM's Rational Rose (Rational Software Corporation 2003), Microsoft Visio (Microsoft Corporation 2007), Telelogic System Architect (Telelogic AB 2007), ArgoUML (Argo/UML), and others:

These tools vary in their exclusivity of support for UML. Tools such as ArgoUML and Rose focus their support almost exclusively on UML, while tools such as Visio and System Architect support diagrams in many notations. All the tools support the creation and manipulation of UML diagrams, generally through a point-and-click graphical interface. The form of this interface—what menu options are available and so on—depends on the

individual tool. UML does not standardize how users are to interact with these diagrams. For example, Rose uses a tree-based view to organize diagrams within the model and property sheets to decorate various model elements.

The tools differ widely in their support for additional capabilities beyond plain UML modeling: design aids, artifact generation, constraint analysis, and so on. Sometimes, third parties will create plug-ins for the tools that add additional features.

A generic diagramming tool such as PowerPoint can be (and often is) used to create UML diagrams, even though it has no specific understanding of UML's (graphical) syntax and semantics. These tools, then, cannot provide any guidance to UML users to help create consistent or complete documents. UML-aware tools, on the other hand, provide design guidance as users create and edit UML. These features include:

Support for UML's built-in extension mechanisms: Recall that UML profiles allow users to specialize elements and introduce additional commonalities among them. UML tools may allow a user to define stereotypes and manage associated tagged values, applying the stereotype's tagged values to all elements of the stereotype, for example. The user interface may also provide new options when stereotypes are applied: The visual appearance of an element may change (adding an icon, for example) and new options may be presented to the user to manipulate the properties associated with that stereotype.

Support for consistency checks: The ability to specify and automatically check constraints can increase stakeholders' confidence in the quality of the architecture. Most UML tools guide the user in making syntactically correct UML documents. The syntactic constraints of UML are generally basic well-formedness checks—that links have valid endpoints, or that two elements do not have the same name, for example.

Many tools go further, and allow the user to check semantic and cross-diagram consistency constraints. For example, a tool may warn a user when a sequence diagram indicates an invocation on an object whose class does not implement a method with the same name as the message. Not all of these constraints are necessarily part of the UML standard, but they are commonly accepted conventions about particular relationships in UML and help to increase the internal consistency of UML models. One danger in this regard is that users of a particular tool may not be able to distinguish easily which constraints are part of UML itself, and which are just being enforced by the local tool. Beyond these generic syntactic and semantic checks, UML includes a constraint language, OCL (Warmer and Kleppe 1998), that allows users to specify constraints on UML elements and element relationships. Tools are available that can assist users in specifying OCL constraints, and then check them automatically. UML does not mandate the use of OCL as its only constraint language, however, so it is possible for UML tools to incorporate additional constraint languages.

Generation of other artifacts: UML models can be used as the basis for generating artifacts useful in other life cycle activities, such as implementation or testing. Some UML tools include facilities for generating these artifacts, or parts of them. For example, Rose has an internal tool called SODA that focuses on the generation of other artifacts from Rose models. SODA effectively transforms UML models into other formats using user-definable templates. Different templates will transform UML

models into different kinds of artifacts: One template may output documentation in a Microsoft Word file; a different template may take the same UML model and generate code skeletons for each of the defined classes. Automatic generation of artifacts that are important in non-design phases of the software engineering life cycle (such as code artifacts for the implementation phase) is especially useful in preventing architectural drift and architectural erosion.

Generation of UML: Some UML tools can take other artifacts (usually implemented systems) and automatically generate partial UML models for them. For example, some tools can iterate over a series of Java classes and automatically generate a corresponding class diagram. These inferences are not always 100 percent correct or useful. They may fail to make inferences about dependencies between classes because a mechanism such as dynamic class-loading or reflection is used. They may also generate every class dependency, when only some of them are architecturally important. For this reason, automatically generated UML needs to be reviewed and “tuned” by hand.

Traceability to other systems: Generation is not the only connection that UML tools may use to interoperate with other artifacts; traceability and links can be used as well. Linking mechanisms may be provided to connect UML elements with system requirements in a requirements management environment such as Telelogic DOORS (Telelogic DOORS), with configuration management repositories, with code-based integrated development environments, with simulation environments, and so on. When such links are available, additional consistency constraints can be checked as well. This can extend the scope of a UML modeling environment beyond UML itself and incorporate other architectural (and nonarchitectural) concerns into the modeling process.

Takeaway Message. UML tools have a profound effect on how users experience modeling in UML. While all tools provide basic support for drawing UML diagrams, the real value in the tools often comes from features provided above and beyond basic drawing: checking additional constraints above and beyond “plain” UML, customizability, generation of other artifacts, reverse engineering code, traceability to other artifacts managed in different environments, and so on. These features generally automate processes that would otherwise be tedious, manual, and error-prone. It is important to be cognizant of any differences between the standard UML language and peculiarities or limitations introduced by specific UML tools. Also, UML tools may lag behind the latest developments in the UML standard itself.

16.2.5 Telelogic System Architect

Telelogic System Architect (formerly Popkin System Architect) (Telelogic AB 2007) is a popular tool among system engineers. It is used to manage diagrams of many different views of a system in multiple diagrammatic styles. It has support for more than fifty different types of diagrams drawn from different modeling methodologies—all the UML diagrams, IDEF, OMT, generic flowcharting, and more. Beyond the basic System Architect tool, a number of variants are also marketed: for DoDAF, for Service Oriented Architectures, for designing enterprise-resource planning (ERP) systems, and so on.

System Architect can support this wide breadth of diagram types because of its internal construction. Internally, diagram data is stored in a relational database that is updated as

the user creates and manipulates diagrams. Each diagram type is associated with a particular vocabulary of symbols and interconnections, as well as constraints that assist the user in creating syntactically correct diagrams.

System Architect is an example of a “standard” tool that trades semantic strength for breadth. It can be viewed as a very good domain-specific diagram editor. That is, it has specific support for drawing diagrams with the syntax of UML, IDEF, OMT, or any number of other types of diagrams. However, it has little understanding of the semantics of these diagrams. The specialized versions of the product (for example, System Architect for DoDAF) have some additional semantic knowledge built in such as the ability to generate or check the consistency of different diagram types.

Takeaway Message. Telelogic System Architect is a de facto standard, flexible drawing tool that gives users a plethora of diagram types in which to express architectural decisions. On one hand, the construction of the tool makes it very easy for Telelogic to incorporate new diagram types and adapt to new diagrammatic languages and standards that emerge. However, this comes at the cost of having more limited associated support for the semantics of those notations. In System Architect, it is certainly easier to draw diagrams with complex syntactic vocabularies than in a completely generic tool like PowerPoint, but it provides little guidance to users as to the quality, correctness, or consistency of their architectures.

16.3 PROCESS STANDARDS

The standards we have covered thus far are mostly related to how to capture an architecture, either conceptually, in a specific notation, or using a particular tool. Process standards dictate the process—that is, the steps—that are used to develop software. These standards focus less on artifacts and more on the methods used to create them.

16.3.1 Rational Unified Process

The Rational Unified Process (RUP) (Kruchten 2003) is a standard that was created by the Rational Software Corporation, now IBM. Fundamentally, the RUP is an iterative process, and is inspired by Barry Boehm’s spiral model (Boehm 1988). Rather than prescribing a single, concrete sequence of steps to follow when developing software, the RUP is an adaptable process *framework*, intended to encompass many different iterative processes and to be tailored for each of them. RUP describes software construction as a whole and is not focused primarily on architectural development (like TOGAF’s ADM).

A RUP project is done in *phases*. The RUP identifies four specific phases:

Inception phase: The inception phase includes initial activities that determine the goals and success criteria for an iteration; it includes making a business case, describing basic use cases, assessing risks, establishing budgets, and so on.

Elaboration phase: Here, use cases are further elaborated and architectural design decisions are made to satisfy them. Business plans and risk assessments made in the inception phase are refined and revisited. Prototypes may be developed.

Construction phase: This phase largely encompasses the implementation and construction of elements that are designed and specified in the elaboration phase.

Transition phase: In this phase, the developed product is transitioned to the end users. It is validated and tested, and its functionality and quality are assessed against the goals set in the inception phase. Deployment, installation, and training are performed in this phase.

The goal of each phase is to reach a *milestone*. For example, the inception phase ends with the "Life Cycle Objective Milestone," which requires stakeholder agreement on scope, cost, and schedule, elaboration of high-level use cases, agreement on credible risk assessments, and so on. Internally, each phase consists of smaller activities apportioned to various personnel. These activities are broken up by *disciplines* such as requirements, design, implementation, testing, and so on. Iterations of these activities may occur within each phase. While the inception phase is expected to be short and is usually done in a single iteration, the other phases often involve multiple iterations.

Figure 16-14 shows the relationship among phases, iterations, and disciplines in a hypothetical instance of RUP. Time proceeds from left to right. Note how the development

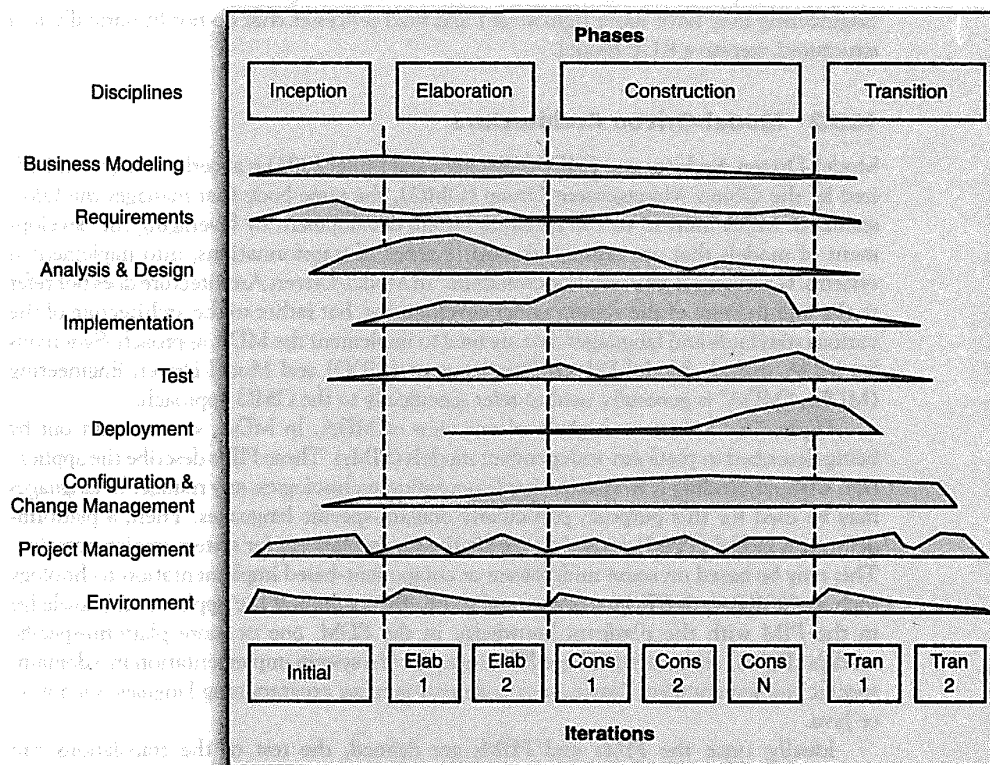


Figure 16-14. Rational Unified Process phases, iterations, and disciplines (adapted from Kroll and Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*).

effort is partitioned by the phases and iterations occur within each phase. The dotted breakpoints between the phases indicate milestones that must be reached. Although all disciplines may be addressed during each phase/iteration, some disciplines are more prominent than others in different phases. For example, very little implementation occurs in the inception phase (perhaps proofs-of-concept or initial prototypes), while a large amount occurs during the construction phase. Likewise, requirements are a heavy focus during inception but are only revised and refined during the construction phase.

Takeaway Message. Ever since the development of the spiral model, software engineers have widely recognized that the use of iteration and risk analysis are important. RUP is a meta-process or process framework in which many iterative (and linear) processes are instances. For example, one could view the traditional waterfall model as a single-iteration instance of RUP. RUP has a tendency to view architecture as a design artifact, rather than as a pervasive discipline—as we have advocated in this book—but the two notions are not entirely incompatible. RUP is not necessarily a universal meta-process; certain development strategies such as Extreme Programming or Domain-Specific Software Engineering may have more lightweight and fluid processes that do not fit naturally in a structured, iterative RUP model.

16.3.2 Model-Driven Architecture

Model-Driven Architecture (MDA) (Mukerji and Miller 2003) is a methodology standardized by the Object Management Group (OMG), the same body that manages the UML standard. MDA aims to reconceptualize system development by leveraging the development of models that are refined, through a series of transformations, into implemented systems. Confusingly, the word “architecture” in Model-Driven Architecture does not refer to the architecture of the system under development, but rather to the architecture of the various standards and languages that are used to implement the MDA approach. Synonyms for MDA include Model-Driven Development (MDD) and Model-Driven Engineering (MDE); “MDA” is generally used to refer specifically to the OMG approach.

Figure 16-15 shows a high-level overview of MDA. In MDA, systems start out by being described in platform-independent models (PIMs). These PIMs describe the application without binding it to specific implementation technologies; any number of languages may be used for this purpose, particularly domain-specific languages. Then, a platform-definition model (PDM) is used to specify the target platform for system implementation. This may be based on some middleware or component-based implementation technology such as CORBA, .NET, JavaBeans, and so on. By combining the application knowledge in the PIM with the platform knowledge in the PDM, one or more platform-specific models (PSMs) are created. These PSMs are part of a system implementation in a domain-specific implementation language or a general-purpose programming language such as C or Java.

Ideally, once the PIMs and PDMs are defined, the rest of the translations can occur using (or at least leveraging) automated translation tools. The separate OMG Queries/Views/Transformations (QVT) standard is one way of defining model transformations. Assuming the translations are correct and preserve salient properties, the result of MDA should be a system that is faithful to the original models.

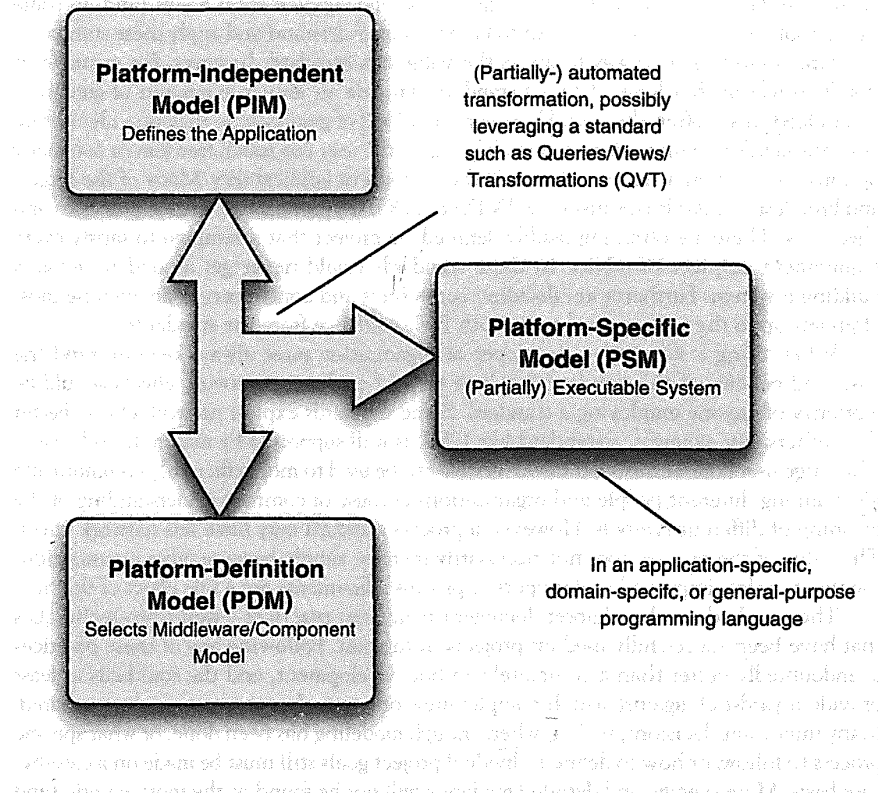


Figure 16-15.
High-level
overview of
MDA.

Takeaway Message. As a general approach, MDA is compatible with many of the mechanisms and ideas espoused in this book. For example, Chapter 9 on implementation discusses how generative technologies can be used to automatically generate (partial) implementations from architectural models, especially when an architecture implementation framework is used. The idea behind MDA is easy to grasp, but the practical details are somewhat more elusive. Making automated transformations from models into implementation artifacts is difficult. Developing completely platform-independent models of an application also is hard to do, since knowledge of available implementation technologies always informs and constrains architectures. MDA is much easier to employ in the context of DSSE, where a well-understood and constrained domain, reference architectures, and existing frameworks provide a much better-elaborated foundation.

16.4 END MATTER

This chapter has attempted to provide high-level overviews of some of the most popular and well-known standards used in practice. One thing that is not evident from this chapter is the level of detail at which many of these standards are specified. Specifications for some

of the standards exceed the length of this book, and supplemental books (and, in some cases, book *series*) have been written to help people understand and apply these standards.

Size is rarely a good way to assess the value of a standard, however. For example, it stands to reason that large, detailed standards provide an immense amount of guidance, and indeed this is often the case. However, what kind of guidance do they provide? Often, large standards remain underspecified: They give the user too much freedom or too many options, often in an attempt to broaden their appeal or applicability. Many of the largest and broadest standards (for instance, DoDAF, TOGAF) prescribe extensive processes and checklists. These are often impossibly detailed—a project that attempted to satisfy every requirement or piece of advice in these standards would never get around to actually building a system. Furthermore, deciding correctness and consistency is an exercise most often left up to the system developers, with little guidance from the standards.

When using a standard, a developer or organization must always keep in mind the costs and benefits of the standard. Looking for and exploiting network effects should be a priority of anyone employing a standard. Some standards exploit network effects better than others. For example, a standard like UML is well supported by mature tools because of its large user community, and UML models can be used to more efficiently communicate ideas among different people and organizations because of common understanding of the meaning of different symbols. However, a process standard may have less network effect: The value of the process does not necessarily increase simply because more organizations use the it, unless improved tool support or process refinements are a side effect of that use.

The standards in this chapter document many best practices—well-worn techniques that have been successfully used on projects in the past. Following any of these practices is undoubtedly better than a completely ad hoc development, and the standards at least provide a yardstick against which completeness or comprehensiveness can be measured. Many important decisions, such as when enough modeling has been done, or what specific process to follow, or how to define individual project goals still must be made on a case-by-case basis. More specific and detailed guidance will not be found in the most generic (and unfortunately, well-known and well-supported) standards. Developers may have more luck combining the breadth of generic standards with more specific domain-specific standards—if such standards exist.

Standards are a big part of using software architecture in practice: Their creation, development, and use are motivated as much by business and organizational goals as technical ones. In the next chapter, we continue the theme of software architecture in practice by focusing on organizational concerns: people, roles, and teams.

The Business Case

From a business perspective, standards can be seen as a make-or-buy decision. Adopting a standard confers many benefits: Adopters are essentially buying the experience and time of previous engineering efforts, vendors, and so on. As those organizations evolve the standard, the quality of the standard is upgraded, effectively for free, and organizations can make decisions on when and whether to move to the latest version of a standard.

Standards provide tremendous value by creating network effects: The more organizations that use the standard, the more valuable the standard becomes. If multiple organizations are involved in a single development effort, standards are often the best way to minimize communication breakdowns and integration difficulties that can kill an otherwise successful project.

When dealing with standards, businesses must keep in mind two primary threats. The first is becoming overly invested in or faithful to a particular approach when that is not warranted. The most reasonable approach is to adopt the standards that make sense, but prepare for a moderate investment in figuring out how to tailor those standards for your purposes.

Many businesses and organizations adopt standards with the mistaken assumption that all their business problems will be addressed if they simply follow the standard to the letter. As we have attempted to convey in this chapter, many of the most well-known, well-supported, and popular standards are often the most general and underspecified. Keeping a rational perspective on the scope of these standards is a must for any development organization. Organizations should define their business goals and the strategies for achieving them independently of any particular standard, and then choose and adapt standards that help them meet their goals based on those strategies.

The second threat is ignoring a standard that eventually becomes popular and finding the standard manipulated by competitors. For this reason, many large companies simply participate in every standardization process related to their business. For organizations that do not have the resources to do this, a more judicious and agile approach is appropriate. These organizations should minimally track the development of all relevant standards, but internally avoid getting too attached to any one standard.

16.5 REVIEW QUESTIONS

1. What is a standard?
2. What is the difference between a de facto and de jure standard?
3. What is a network effect? How do standards help to create a network effect?
4. What are some drawbacks of standards?
5. What are the advantages and disadvantages of adopting a standard early? What are the advantages and disadvantages of adopting a standard late?
6. What is the scope of IEEE 1471? How does it advise and guide users?
7. Describe DoDAF. How does DoDAF help its users? What does DoDAF not provide?
8. What is TOGAF and how does it differ from DoDAF? What is the difference between enterprise architectures and software architectures, as they have been described in this book?
9. What are the five RM-ODP viewpoints? What kinds of systems is RM-ODP targeted for?
10. What are UML profiles and how do they help users in applying the UML standard?
11. What is SysML and how does it differ from UML?
12. What lessons can be learned from the standardization efforts behind ADML and AADL?
13. Enumerate some standard architecture tools. How do standard tools augment standard notations? How can they hinder the use of a standard notation?
14. Enumerate some process standards. How do process standards augment and interact with other standards?

16.6 EXERCISES

1. Obtain and read one of the standards listed in this chapter. Choose a system with which you are familiar (or one of the ones used as an example in this book, such as Lunar Lander or the World Wide Web) and outline how it would be described or implemented using that standard. Identify the standard's strengths and weaknesses in the context of your experience.
2. Draw out and discuss the relationships between the standards discussed in this chapter. Are they mutually exclusive? Do they dovetail with one another? Are they sequential? Identify some software development standards not discussed in this chapter and add them to your diagram.

3. Find an architectural model that claims to conform to one of the standards in this chapter. Check it for conformance against the standard itself.
4. Choose a standard and investigate how it is described and marketed by its creators and others. Assess whether the benefits (and drawbacks) claimed of the standard are accurate or not.
5. Contact one or more software development practitioners and find out about their use of standards. Ask what benefits they believe they are deriving from the use of the standard, and assess whether their development experiences have borne these out.
6. Find a tool set that purports to support a particular standard. Select various elements of the standard and determine how they are supported by the tool set. Identify how the tools help or hinder the use of the standard.

16.7 FURTHER READING

This chapter attempts to distill and extract the essence of some of the most well-known and influential standards in software engineering today. Still, the few pages dedicated to each standard cannot capture the scope of most of them. As noted above, the length of several of these standards exceeds the length of this book.

The best way to learn about a standard is often to go to the source. References are included in the bibliography for IEEE 1471 (IEEE-SA Standards Board 2000), DoDAF (DoD Architecture Framework Working Group 2004), TOGAF (The Open Group 2003), RM-ODP (ISO/IEC 1998), UML (Booch, Rumbaugh, and Jacobson 2005) and its meta-model the MOF (Object Management Group 2002), SysML (SysML Partners 2005), AADL (Feiler, Lewis, and Corporation 2003), ADML

(Spencer 2000), Rational Rose (Rational Software Corporation 2003), Telelogic System Architect (Telelogic AB 2007), RUP (Kruchten 2003), MDA (Mukerji and Miller 2003), and others. Beware of the “executive summaries” contained in some of these standards or related documentation; these descriptions are often heavy with hype and it is difficult to get a clear assessment of a standard from them. For comparative purposes, it might be useful to examine standards not related to architecture such as POSIX (IEEE 2004) and HTTP (Fielding et al. 1999) or domain-specific standards such as the Software Communications Architecture (SCA) (Modular Software-programmable Radio Consortium 2001), introduced in Chapter 15.

People, Roles, and Teams

It has become quite common for a software engineer to be granted a title such as “software architect,” “senior software architect,” or “chief systems architect” in a software development organization. Such a title appears to carry with it a certain amount of prestige. The usual assumption is that the engineer in question has exhibited exceptional skills in software system design, familiarity with the modern development technologies, an ability to communicate a vision for a project, effectively interact with system customers and managers, and lead his fellow engineers in completing the project successfully. At the same time, it is likely that someone looking a bit further will find many different descriptions of what software architects actually do from one company to another, and even from one project to another within a single company. In some companies, “chief architects” are brilliant engineers who worked in the trenches for many years and emerged as technically savvy sages; in other cases, they are former CEOs who decide to step down and shift their focus to ensuring their company’s marketplace presence and setting the company’s long-term technical strategy. What this means is that, in fact, simply having the title does not mean one has the qualifications.

Clearly, this is very different from, say, an electrical engineer or even a software engineer: Becoming one usually requires completing at a minimum an appropriate four-year college degree, and perhaps a further number of years of practical experience with specific additional training. By contrast, it is not unusual that a person with “software architect” in his title is not even a software engineer by training. This carries with it a significant danger: Identifying, nurturing, and promoting a software architect becomes similar to building an underspecified software system. In software development this is sometimes referred to as the IKIWISI syndrome, or “I’ll know it when I see it.” Discovering the right software system, just like discovering a good software architect, under such conditions can be too time consuming, too risky, too unpredictable, and too expensive. Moreover, it is also unclear how one can go about becoming an architect: The title tends to be bestowed upon, or sometimes assumed by, a person rather than being earned through a well-understood set of steps. And should that person move to

a different organization, or even a different division within the same organization, the same set of job responsibilities might carry a completely different title. Finally, with very few exceptions, one will not find a training program within a (future) development organization that helps an individual get that job or an appropriate title.

If our goal is to study software architecture as a mature discipline with well-understood and codified objectives, principles, and practices, then we must define more clearly the roles and responsibilities of the discipline's practitioners, namely, software architects. This chapter will do just that. We are guided by the observation that software engineering is a multifaceted discipline that combines:

- *Core engineering skills*—for example, formal system modeling, code production, and product testing and measurement.
- *Organizational skills*—for example, project management, cost estimation, and adherence to industrial and government standards.
- *Interpersonal skills*—for example, communication with many stakeholders, honoring one's roles and obligations within a team, and putting the project's success before one's own personal goals.

Software architecture permeates the entire software engineering process, and in many ways forms its core; it is not unreasonable to require that software architects possess a large cross-section of these skills. The goal of this chapter is to answer four key questions:

1. Who are software architects?
2. What do software architects do?
3. How do they go about doing their job?
4. How do architects relate to, and interact with, other stakeholders in designing, implementing, and evolving a system's architecture?

The first question deals with the core set of skills a software architect must possess, including both technical and nontechnical skills. The second question sheds light on what a software architect's job actually entails. The third question addresses how software architects are organized into teams, and how and when they move from one project to another. Finally, the last question takes a broader perspective on software architecture in practice as an exercise in consensus building in order to satisfy all of a system's stakeholders. We will address each of these questions in more detail.

17 People, Roles, and Teams	17.3 How Do Software Architects Work?
17.1 Who Are Software Architects?	17.3.1 Balance of Skills
17.1.1 Architect as a Software Designer	17.3.2 Allegiance to the Project
17.1.2 Architect as a Domain Expert	17.3.3 Allegiance to the Organization
17.1.3 Architect as a Software Technologist	17.3.4 Duration of Involvement
17.1.4 Architect as a Standards Compliance Expert	17.3.5 Team Structure
17.1.5 Architect as a Software Engineering Economist	17.4 How Do Software Architects Relate to Other Stakeholders?
17.1.6 Some Bad Habits	17.4.1 Architects and Engineers
17.2 What Do Software Architects Do?	17.4.2 Architects and Managers
17.2.1 Develop Project Strategy	17.4.3 Other Stakeholders
17.2.2 Design Systems	17.5 Remaining Challenges
17.2.3 Communicate with Stakeholders	17.6 End Matter
17.2.4 Lead	17.7 Review Questions
	17.8 Further Reading

Outline of Chapter 17

17.1 WHO ARE SOFTWARE ARCHITECTS?

It has been said that software architecture is a result of applying “method, theft, and intuition” (Kruchten 1995).

- *Method* refers to a well understood, codified set of steps that can be repeatedly executed to solve a particular problem, or category of problem.
- *Theft* refers to the reuse of solutions that have been shown successful in the past.
- *Intuition* is the ability to conceive, understand, and apply ideas without necessarily being able to communicate or explain the entire rationale behind them.

In fact, the reader has seen different elements of method and “theft”—that is, reuse—discussed throughout this book. These include reliance on effective techniques, processes, and existing solutions, such as styles, patterns, and entire DSSAs.

In many ways, the essence of a software architect's job is to be able to apply and find the right balance among these three “sources” of architecture. In addition, by definition an architect is given a great deal of responsibility on a project; the project will succeed or fail depending on the architectural solutions employed. In many software development organizations that responsibility is not accompanied with specific authority. Instead, the architect is usually considered by the management as one of the engineers or, at best, as “first among equals.” In order to do his job effectively, the architect must then find another way to exert influence and lead. He must possess and rely on a number of skills. An architect must be:

1. A software designer.
2. A domain expert.
3. A technologist.

4. A standards compliance expert.
5. A software engineering economist.

17.1.1 Architect as a Software Designer

First and foremost, a software architect must be an excellent designer of software systems. He must be able to recognize, reuse, or invent effective design solutions and apply them appropriately. He must be familiar with the key architectural styles and patterns that underlie the discipline of software architecture. An architect may even have some patterns and/or styles that are part of his private arsenal of design tools, amassed over time, possibly as a result of work within a specific domain or application family. Conversely, an architect must also be able to recognize ineffective or suboptimal designs, inappropriate styles and/or patterns, and design decisions “with strings attached.”

In an ideal world, a good architect would always be able to provide rationale for all his decisions. However, we should also recognize that truly great architects—software or otherwise—often leave their mark when they depart from the conventional wisdom, for example, when they are faced with an unprecedented problem, or when they realize that an existing problem can be solved in a better way. Their inspiration in such cases may be difficult to rationalize.

To be a great designer of software systems, a software architect must be a skilled software or systems engineer, with years of experience that is both rich and varied, good communication skills, and a keen sense of aesthetics. A software architect must be an engineer because ultimately he is solving engineering problems. More specifically, he must have training as a *software* engineer because software systems exhibit many properties not shared by other types of engineered systems.

A good architect must have extensive experience because he must strive for the most effective balance of method, theft, and intuition for the problem at hand. For example, if he is dealing with an unprecedented system, he might need to rely on intuition honed by working on many different types of problem over the years and apply generic methods, such as object-orientation. On the other hand, if he is dealing with a variation on a known problem, he will be able to leverage directly his experience, and apply both specific methods and reuse solutions from prior systems.

The architect must also be a good communicator since he must convince many other people of his vision for the system. This includes, for example, other engineers who actually will build the system, but may also include the managers, external project collaborators, project customers and users, venture capitalists, and perhaps even the company's board of directors.

Finally, the architect must have a good appreciation for aesthetics. Those who cultivate such appreciation of design in the real world do a better job of designing in the software world. This point is only briefly stressed here, but has been discussed throughout the book (for example, see Chapters 1, 4, and 11). Elegant designs promote excitement about a project among its stakeholders, including the developers. They are remembered, studied, and emulated. In contrast, boring designs are not.

17.1.2 Architect as a Domain Expert

Many software development organizations produce multiple systems within the same application domain. For example, Microsoft works primarily within the domain of

desktop applications; Google's primary domain of interest is information dissemination and use over the Internet; companies such as Boeing and Lockheed Martin have an extensive focus on embedded systems; NASA produces ground-, Earth orbit-, and space-based systems.

In some cases, the domain of focus will be accompanied by one or more domain-specific software architectures (DSSAs) (recall Chapter 15) that codify the characteristics of the problem being tackled as well as the solution developed in response. The burden of intimate familiarity with the application domain and the greatest risks of inappropriate and ineffective solutions are removed from the architects in such systems; they need only familiarize themselves with and stick to the prescription contained in the DSSA.

However, the systems being developed in some companies will be related only loosely, such that they cannot be consolidated into a DSSA. In those cases, each system creates its own challenges. The software architects working on such systems will need to have an intimate understanding of each application domain, its major properties, and its idiosyncrasies. Otherwise, regardless of their skills as designers, architects are likely to produce designs that do not work well for the type of task at hand. They may have problems communicating with other stakeholders, including other engineers. And they may not be able to use the properties of the domain to their advantage when facing a problem with their architecture.

17.1.3 Architect as a Software Technologist

A software architect must know that his solutions actually will work once fully developed. Just like a building architect needs to ensure that his designs can be constructed, so a software architect needs to ensure that the software technology exists to support his ideas. A beautiful design is useless if it cannot be implemented.

An architect's primary role, and likely even his main strength, is not as a programmer. Indeed, experience has shown that outstanding programmers do not necessarily make outstanding architects. At the same time, the architect cannot divorce himself completely from programming. The reasons behind this are analogous to the arguments made throughout this book for why it is unrealistic to isolate completely a system's architecture from its implementation. Architects must progressively elaborate design concepts into concrete elements. To do so, they may need to prototype specific elements of their solutions to ensure that those elements will behave in practice as envisioned. They may need to test out ideas on the exact execution platform on which the system will run. The architects may also need to convince either managers or developers that a particularly controversial idea is worth pursuing, by showcasing its benefits in practice.

Architects also must have a grasp of the generic capabilities software developers will have at their disposal when implementing the architecture. In the previous chapters we have covered many such technologies and shown that they can have a direct bearing on a system's architecture. This may include the details of the operating systems and programming languages on which the eventual application will run. It may also require the architect to familiarize himself with the latest advances in software libraries, frameworks, middleware platforms, networking solutions, and so forth. The architect's job is not only to ensure that his ideas can be implemented; it is to ensure that they can be implemented effectively and that the process of creating that implementation is efficient.

Additionally, the architect may need to play the technologist card because of perceptions. Due to the nature of the job, the architect must project authority with different stakeholders. Since an architect often is not a manager on whom authority is explicitly bestowed, that authority must be earned. This is especially the case in the architect's interactions with engineers who have no competence in software development, yet who may be the architect's most important clients. Engineers are typically highly educated and skilled themselves, very protective of their work, and may believe that they are capable of solving a particular problem just as well as the architect—regardless of their lack of any genuine qualifications in software. Being able on occasion to demonstrate technical prowess will help the architect remind the engineers that he is one of them, of the reasons he holds the job, and that he does not live in the proverbial ivory tower.

17.1.4 Architect as a Standards Compliance Expert

Many government, legal, and technical bodies produce standards to which a development organization must adhere. Some of those standards are mandated externally and are the precondition for bidding on contracts. For example, many projects funded by the U.S. Department of Defense must adhere to the C4ISR architectural framework. Other standards may be required by the company's management. For example, a company may decide to use UML 2 as an architecture documentation tool.

In either case, a software architect must be intimately familiar with the relevant standards. They must be able to accurately assess and communicate the value of specific standards, or the lack of appropriate standards. They must understand a given standard's expected impact on the architecture and on the eventual product—both positive and negative—and be able to explain that impact to other stakeholders. Finally, they must be able to demonstrate that their architectural solutions adhere to any chosen standards, for example, in the case of a project audit.

The architect should also follow newly initiated standards efforts. A standards body may produce a technology that will significantly simplify development of future products, improve modeling and analysis capabilities for existing architectures, or ensure easy interoperability with important third-party software. For example, the emerging Architectural Analysis and Design Language (AADL), which was discussed in Chapter 6, may provide modeling capabilities that UML does not possess; a given extension to the CORBA standard may ease system implementation; and so on. Additionally, actual participation in standardization efforts may help prevent ill-conceived and resource-wasting standards from being imposed on future projects and software architects.

Standards Compliance Roadblocks

Despite his best efforts, a software architect may at times be unable to ensure full compliance of a given architecture with a given standard. There are at least two possible reasons for this.

First, the standard may be loosely defined. It may provide a set of general guidelines, rather than a rigorously specified notation, technique, or process that an organization can follow and use. An example of a loosely defined standard is DoDAF, which was discussed in the preceding chapter. Another example is UML itself: While its syntax is precisely defined, its semantics is not.

The second reason a standard may be hard to comply with is that the tools built to support the standard may, in fact, deviate from it. For example, a leading UML-based software modeling environment is IBM/Rational Rose. However, throughout UML's existence and evolution

since the late 1990s, Rose has tended to deviate from the Object Management Group (OMG) standard, even if only in subtle ways. In such a situation, an organization that purchased Rose for modeling its software systems would be induced to deviate from standard UML.

Of course, the opposite of this argument also holds: These very reasons make it difficult to argue that an architecture does not comply with a standard.

17.1.5 Architect as a Software Engineering Economist

A great software designer is not necessarily a great software architect. His designs may indeed be superior when looked at in isolation, but in the context of a particular organization or project, those designs may be unrealistic, too expensive, technologically too ambitious, in violation of certain technological agreements or legal codes, or too risky for a multitude of other reasons. In other words, an architect must be more than just a (great) designer. An effective architect must produce an architecture that effectively addresses the problem at hand while simultaneously respecting the constraints placed on the project. If it fails on either count, that architecture will very likely result in a failed project.

One of the frequent constraints in real-world projects is money. Some companies competitively bid for contracts whose size constrains what can be done in the course of the project. Examples of such companies are large enterprises that work on government-funded projects in the United States and Europe: Lockheed Martin, Boeing, BAE Systems, and so on. Other companies get a finite, frequently quite small amount of venture capital to produce a successful system. Examples include many dot.com start-ups. Yet other companies go to their own coffers to fund a project that is expected eventually to turn a profit. Such companies include large commercial software outfits, such as Microsoft, Yahoo, and Google.

In each case, when producing an architecture for a system under development, the architect must not only be aware of the monetary constraint, but must also be able to determine whether his solution can be implemented effectively within the project's budget. This means that the architect must have some, at least basic, understanding of the economics of software development, and must accompany his design decisions with, possibly quick and crude "back of the envelope" calculations of how much their implementation is likely to cost. Budgetary considerations may also lead the architect to opt for adopting or adapting existing architectures and for going with off-the-shelf functionality. Being able to provide appropriate economic justification for all such decisions is an integral part of the architect's job. Ultimately, the architect must be constantly aware that the solution he produces may very well be technically suboptimal.

17.1.6 Some Bad Habits

There are also certain tendencies an architect must avoid. Some obvious bad habits are discussed here; the reader may be able to identify others as well.

Perfectionism. The constraints inherent in the architect's job often will make it impossible to produce the perfect architecture for the given problem. Among the many reasons discussed throughout this text, at any given time the architect is likely to possess only partial information. That information may change over time such that, as the architect discovers new details about the system he is working on, previously held beliefs become invalid.

Striving for perfection in such a fluid environment is impractical. In fact, one's ability to acknowledge, respond to, and even embrace change will be a mark of one's success as a software architect. This leads to the second bad habit.

Inflexibility. An architect should avoid insisting on a single correct or best way of constructing a system. A basic, positive, tendency of a designer should be to produce a clean and simple design. When constructing a building, bridge, or dam, this may mean clearing the building site so that the site fully reflects the architects' and engineers' vision. Unfortunately, it does not always work that way with software. Very few large systems are built from scratch, using a single architectural style, one's favorite set of patterns, standard middleware, programming language, operating system, and hardware platform. Instead, a software architect is much more likely to have to mix and even invent styles and patterns, look for new middleware platforms that solve the new problem more effectively, address multilingual development on multiple platforms, and so on. The reasons for this may be legitimate technical considerations. The reasons may also be the many conflicting and changing requirements levied on the architecture and the architect by the other system stakeholders. In many ways, a big element of a software architect's job is deciding which stakeholders to disappoint.

Micromanagement. Another bad habit for a software architect is the impulse to micromanage. An architect produces designs that may result in huge systems developed by many teams, possibly from many organizations, over a significant time period. Making sure that everything goes perfectly and that everyone adheres to the architect's vision at all points in time is simply not possible. It is also not the architect's job. A good architect will recognize that he is surrounded by highly skilled engineers and that many, or even most, of them will be able to do their own jobs much better than he will. Furthermore, trying to meddle in their jobs may create resentment and undermine the architect's goal of leading by example.

Isolationism. Finally, the architect should not isolate himself. Even though the architect may feel that the weight of a project is on his shoulders and that, at least early on in the project, he bears the greatest burden, the architect must remember that he is part of a much larger team, and that an integral part of his job is to ask, watch, listen, and communicate. The architect may get solutions to his specific problems in the process. He will also get the early buy-in from the stakeholders, a sense of joint ownership of the architecture, and the chance to set the stage for the architecture's eventual implementation. Shutting oneself in a room because one is "the architect" will likely achieve the exact opposite effect.

Another aspect of isolating oneself is insisting on always devising and using one's own solutions. Even though at times there may be a legitimate need for an architect to invent new design techniques, notations, patterns, or styles, the accompanying danger is doing this even when it is not necessary. This is often referred to as the "Not invented here" syndrome: Architects and, more broadly, organizations are sometimes unaware of, or even deliberately ignore, known effective solutions and insist on developing their own instead. Closely related to this is the "We have always done it this way" syndrome, where suboptimal or even ineffective solutions are applied repeatedly because they may have been demonstrated to work nominally in the past (possibly in very different contexts and on very different problems). Both of these situations are risky, expensive, and frequently

result in duplicating the work someone else has already done or, worse, producing inferior solutions.

In many software development organizations, especially start-ups, the chief architect is the person who came up with the original idea upon which the company was founded. That person may hold other titles and responsibilities in such an organization, such as "Chief Technical Officer" or even "Chief Executive Officer." Alternatively, a new owner of an existing company may assume the role of the chief architect of the company's flagship product or its entire line of products.

The Ultimate Bad Habit

The responsibility placed on such individuals clearly is greater than it is on "mere" architects: They are responsible for much more than just the technical success of a product. At the same time, these individuals' prior successes, especially if they were big successes—think, for example, Netscape—can become an impediment to doing smart business (and smart architecture). The relatively brief history of the software marketplace teaches us that the fortunes of a product, and of a company, can change rapidly. Therefore, delusions of grandeur is the ultimate bad habit for software architects.

In his fun, insightful, and sobering book, *In Search of Stupidity* (Chapman 2003), Merrill R. (Rick) Chapman chronicles several examples of this ultimate bad habit of software architects from the past few decades. He explains how and why certain companies—including aforementioned Netscape—have gone from being market leaders to historical footnotes through an often rapid succession of missteps, both managerial and architectural. Chapman's book serves as a useful reminder of the need for careful planning, good software architecture, solid engineering practices, and, above all, humility.

17.2 WHAT DO SOFTWARE ARCHITECTS DO?

A software architect's job will require showcasing a number of diverse skills. Being an effective designer is one such skill, but an architect will spend much of his time doing other things. There are four tasks an architect will most likely have to perform regardless of the type of project, his level of experience, the type of organization, the application domain, or the industry segment in which he works. These tasks are developing project strategy, designing systems, communicating with stakeholders, and being a leader. In accomplishing these tasks, the architect repeatedly will have to apply, combine, and extend the skill set described in the preceding section.

17.2.1 Develop Project Strategy

A talented software designer will most often be able to produce an excellent technical solution for the problem at hand. That solution, however, may not be the best one for the *project* or the *organization* solving the problem. A talented software architect will have to recognize the distinction between the two, and make sure always to address the latter. The architect's primary job is to help develop a comprehensive strategy for the project. That strategy, if appropriately implemented, not only will result in a system that is a good technical solution, but also one that becomes a valuable product for the organization.

This requires understanding the economic context of the project and the organization, the competition in the marketplace, and any standards and regulations by which the

organization must abide. Perhaps most importantly, the architect must be intimately familiar with the human as well as technical resources at the project's disposal. An otherwise outstanding design will have a high likelihood of failure if it requires purchasing a lot of expensive new technology, retraining the existing engineers, or hiring many new staff.

17.2.2 Design Systems

A central part of overall project strategy is design of the system's architecture. Once an architect is aware of the relevant constraints being faced, he is ready to proceed with solving the problem. This is likely to be the most enjoyable part of architect's job; paradoxically, it may also be the most stressful.

An architect will likely enjoy designing systems because he will get the opportunity to design, use his imagination and creativity, put his stamp on the future systems, and perhaps even to think outside of the proverbial box. However, the architect will still need to be in frequent contact with other stakeholders, gather information from them, and respond to their possibly conflicting requests. He will have to deal with the likelihood that he may not have the complete picture at his disposal and that the picture constantly changes. It has been said that the job of a software architect is a succession of suboptimal decisions made in semidarkness. The architect will also have to avoid letting personal goals, such as producing a clean design in record time, override the other stakeholders' concerns.

17.2.3 Communicate with Stakeholders

Throughout the process of developing the project strategy and producing the architectural design, the architect will be in frequent contact with the system's various stakeholders, which may include developers, testers, technical leads, managers at all levels, customers, and users. One of the main factors in the project's success is whether the architect can project and ensure a continued, coherent vision for the resulting system. In that sense, his job has something in common with that of a salesperson, with the added twist that he is negotiating with and "selling" the project to many different types of customers in many different ways. The architect may need to interact with the various stakeholders repeatedly, especially in a long-lived project, where the overall vision may be temporarily lost in the details. In such situations, the architect may also need to assume the role of the project's cheerleader, maintaining the morale of the different stakeholders. He may also need to show significant awareness and political savvy regarding the competing interests of the various stakeholders in order to carry out this task effectively.

While the architect may use different languages to communicate with different stakeholders and may focus on different aspects of the project depending on the situation, throughout this process he must ensure the architectural integrity of the project. The compromises the architect must inevitably make cannot be allowed to undermine the overall objective or to clash with earlier decisions made for equally valid reasons. Architectural degradation must be avoided throughout this task.

17.2.4 Lead

An architect makes many decisions that are crucial to the project's success and has to convince a broad audience—a part of which that may be skeptical—that those decisions

are correct. Some decisions may be unpopular, yet the architect needs to get the buy-in from all the major stakeholders, and must do so continuously throughout the project's life span. In some situations, just getting buy-in may not be good enough; the architect may need to elicit excitement for the project. He will frequently have to assume this responsibility without any explicit authority over the stakeholders.

To do this, the architect must be a leader. While it may be difficult to teach, leadership is a skill that can be nourished and refined. To be an effective leader, an architect must:

1. Show confidence in the project's success.
2. Project conviction in his ideas.
3. Demonstrate readiness to assume full responsibility for any technical problems.
4. Be ready to articulate the technical rationale for design decisions.
5. Be able to further develop the project's detailed architecture.
6. Acknowledge contributions of others.
7. Avoid self-aggrandizement.

At all times, the architect must put the interests of the project before his own, and lead by example. Developing a complex software-intensive system may require sacrifices from engineers, including long hours, working weekends and holidays, and time away from families. An architect who is a true leader will always be prepared to make his share of the sacrifices. He must also know when making such sacrifices is inappropriate, and should ensure that project personnel are treated with respect and consideration.

17.3 HOW DO SOFTWARE ARCHITECTS WORK?

An architect may work alone or, more likely, as part of a team. The exact composition of an architecture team and its precise role during the various project stages may vary from one organization to another, and even from one project to another. However, teams should be built with certain generally applicable guidelines and observations in mind, including a balance of skills, allegiance to the project and the organization, and ability to serve for the duration of the project.

17.3.1 Balance of Skills

The software architecture team must be composed from people whose strengths are complementary and cover the range of skills discussed above. It is unlikely that a single person will be an outstanding designer, domain expert, technologist, communicator, *and* leader. More likely, each architect's primary strengths will be in a small number—perhaps only one—of these areas.

Composing the architecture team from people with similar skill sets thus is not helpful. For one, if certain project needs, such as design, are overstaffed, there is a danger that the architects will get in each other's way, while other aspects of the architecture, such as domain expertise, may suffer. At the other extreme, staffing the team with architects whose skills do not sufficiently overlap carries the danger that the information gathered and architectural decisions made will neither be properly understood by different team members nor integrated into a coherent whole.

The architecture team carries a great deal of responsibility for and exerts dominant influence on a project. Thus, it must be staffed with qualified people. A long-time software architecture practitioner and researcher, Phillippe Kruchten makes the interesting observation that the architecture team should not become a "retirement home" for aging software engineers (Kruchten 1999). This may seem to be an unnecessary admonition. However, it is not uncommon for engineers with outdated technical skills in large organizations to be "promoted" to software architects. This may be seen as the easiest way of minimizing the negative impact of such engineers' deteriorated abilities, but can be very dangerous and, simply put, it is patently stupid.

The Peter Principle

A failure by an organization to staff its software architecture team with competent people can be viewed through the prism of *The Peter Principle*. This principle was formulated by Laurence J. Peter in his 1969 book of the same name (Peter and Hull 1969), and states that, "In a hierarchy every employee tends to rise to his level of incompetence." In other words, a member of a hierarchical organization eventually is promoted to his highest level of competence; any subsequent promotion will elevate the employee to a position for which he is no longer competent, that is, to his "level of incompetence." Once he reaches his level of incompetence, the employee's chances of further promotion are low. However, from a practical perspective, by that time the damage to the organization may already have been done.

For example, an excellent C programmer within an organization may receive many salary raises and may progress through the company's hierarchy of software engineering positions: "software programmer," "software engineer," "senior software engineer," "principal scientist," and so on. At some point, for reasons discussed in this chapter, the employee may be promoted to a position of "software architect," for which he may be poorly qualified. Such a position is likely too important and carries too much responsibility for the organization to be able to afford letting this employee take the time to learn his new job. However, the organization may have no other choice, unless it wants to demote its otherwise excellent engineer. In the meantime, the likelihood of failed projects will be elevated.

The dire implication of the Peter Principle is that an organization is prone to collapse if a sufficient number of its employees reach their levels of incompetence.

17.3.2 Allegiance to the Project

The software architecture team should be an integral part of the project, and should be associated with the project throughout its life cycle. Both the architecture team and the other stakeholders must understand that the architecture team is invested in the success of the project. To the architects, this is important because of the architecture team's need to exert influence over all principal design decisions whenever required. To the programmers, it is important because they must feel that the software architects are "one of us." To the managers and customers, it is important because they can clearly see accountability for the architectural decisions made.

An alternative to this view is to treat the software architecture team as a consulting entity that jumps in depending on the needs of different projects. This can be dangerous for several reasons. If a project experiences difficulties, an "us versus them" division may result. Furthermore, the implicit lack of accountability and oversight by management in that case makes it less likely to get buy-in from the stakeholders. For example, if developers believe that it is their responsibility alone to build a successful system and that the architecture

team is a marginally interested third party, they will be less likely to take and implement the architecture team's advice, thus possibly violating critical architectural decisions. Most importantly, it is highly unlikely that a part-time software architecture team can simply show up whenever needed, understand all the nuances of the system, make appropriate decisions, and move on.

We should recognize the tension that inevitably will appear between the need of the project to have a readily available software architecture team and the need of the organization to leverage the most experienced, talented, and effective architects across multiple projects that may be undertaken simultaneously. There is no easy solution to this problem, and different organizations may try to address it in different ways. For example, more experienced, senior architects may move to another project once a given system's architecture is reasonably stable, while the junior architects would remain with the project, assume additional responsibilities, and further hone their skills.

17.3.3 Allegiance to the Organization

Google the phrase "software architect," and a number of Web sites belonging to consulting firms will appear. These firms specialize in providing software architecture "services" to other software development firms. Their mode of operation is to come into an organization, familiarize themselves with the project and possibly the problem domain, and help develop the appropriate architecture for the system under development. At first blush, this may seem like the part-time software architecture team pitfall mentioned above. However, these teams may be hired to staff the project throughout its duration, alleviating that particular concern.

The real question in this case becomes whether it is preferable to employ a permanent software architecture team that will be part of, and thus fully accountable to, the software development organization, or to outsource this important, but possibly scarce and expensive, capability. The conventional wisdom would suggest that having a native software architecture team is preferable, a stance that probably is influenced by the traditional view of software development in which an organization owns a project and develops it completely in-house. Such organizations and projects still exist in some cases, an obvious example being Microsoft, and this view may be appropriate for them.

The answer to the question of whether to employ or hire a software architecture team becomes more complex in the case of many other development organizations. The reasons are two-fold. First, a typical software project will use a number of off-the-shelf technologies and even application-level components. In such situations, the question becomes, who actually owns and controls the software architecture? It has been shown that such technologies directly impact a system's architecture (Di Nitto and Rosenblum 1999). Therefore, by importing an off-the-shelf capability, an organization, at least in part, imports an architecture as well, and the perception may be that there is less of a need for an in-house architecture team. This may indeed be the case in certain situations. But beware: The danger is that by letting a chosen third-party's technology control the architecture, the organization is, in fact, relinquishing a critical piece of control over the project. And that control is given to people who:

- Did not have that project in mind when they developed their technology.
- May not have taken into account, or even been familiar with, any aspects of the problem domain.

- Are unfamiliar with the organization, the skills of its engineers, the technical and economic context of the project being undertaken, and so on.

In effect, doing this is akin to letting the tail wag the dog.

The second consideration in deciding whether to maintain an internal architecture team has to do with the software development model that has gained increased popularity: outsourcing. Usually a development organization will find it economically advantageous to hire outside developers, often in a foreign country, to implement parts of a system. It might be argued that, by extension, the architecture of the system can be outsourced as well. On the surface, this is similar to hiring an external architecture team. However, relying on an external architecture team assumes that the team would still need to understand the details of the project and its requirements, work in close collaboration with the other system stakeholders, get their buy-in, and oversee the architecture's implementation and evolution. Simply purchasing a putative architecture for a project from a third party usually is not feasible, unless the organization is branching out into a new business area having a well-established existing architecture.

One potentially positive aspect of hiring an external software architecture team or the outright outsourcing of the architecture is that it might be an effective way of avoiding both the "Not invented here," and the "We have always done it this way issues." Even here, however, this risk can never be completely eliminated: The "here" and the "we" may just be shifted to the external team.

17.3.4 Duration of Involvement

The software architecture team does not disband and/or disassociate itself with the project after the architectural design activity is over. There might be an impulse to do so as the team's task is considered to have been completed and other projects may need architectural help. However, moving the team to other projects will have a similar effect to that of treating the team as a consulting entity. More importantly, we have stressed several times throughout this book that it is inappropriate to think of software architecture as a "phase" that is completed early on in a project. While most of the major architectural decisions may be undertaken during the project's early stages, the architecture remains an active, almost organic underpinning for the project that constantly changes and evolves. Cutting off the architecture team from the project can result in many critical architectural decisions being made haphazardly, understood partially at best, and documented poorly, if at all.

Instead, the architecture team must be closely involved with the project throughout the project's life cycle. Once the architecture becomes reasonably stable, it may in fact be possible, and even advisable, to reassign some members of the architecture team to other projects. However, at least a part of the architecture team should remain on the project. In certain cases, members of the architecture team may be assigned as liaisons to the development teams; in other cases, they may become leads of different development teams. This allows the architecture team to have direct oversight of the project's progress, any difficulties encountered, and adherence to the architecture and any needed changes to it. It also helps reinforce the fact that the architecture team's primary allegiance is to the project, thus helping to improve the morale.

17.3.5 Team Structure

Software architecture teams can be structured in different ways, depending on the size of the software development organization, the number and types of projects on which the organization is working concurrently, and the organization's own management structure. We briefly discuss three primary ways of structuring an architecture team. Variations on these models are possible.

Flat Model

The flat model usually works well in small organizations. Architects in it are not stratified in any way and carry comparable clout and responsibility. It is an egalitarian organization that comports well with mature professionals. On the other hand, with less mature or responsible individuals the flat model may cause problems in cases of disagreements, whether within the architecture team or with other stakeholders. It may also overwhelm the architects because of the unspecified or inadequate division of responsibilities. This model may not scale well to large teams and large projects. An extreme case of a flat architecture team model is that of a project staffed by a single software architect.

Hierarchical Model

A more practical model, especially in larger organizations is the hierarchical model. This model was identified as early as Fred Brooks's *The Mythical Man-Month* book (Brooks 1995). In this model, the architecture team is typically headed by a chief architect or senior architect, and staffed with junior architects. In some, typically large, organizations the architecture team may be further stratified. For example, an enterprise architect may be in charge of the software architectures of all of that organization's different projects. The division of responsibilities and authority in the hierarchical model is clearly defined. This makes it more easily scalable to large teams. The potential downside is that the architecture team must in effect assume an internal management structure, in addition to and separate from that in the rest of the organization.

Matrix Model

In the matrix model, architects simultaneously work on multiple projects. A cross-section of skills is applied to a cross-section of projects. A single project may have multiple, changing architecture teams during its life span, depending on the expertise needed. This model can be found in organizations that are understaffed (perhaps unexpectedly) or experience periodic spikes in undertaken, externally funded projects. The matrix model is orthogonal to both the flat and hierarchical models: Architecture teams may, but need not be, stratified. The main advantage of this model is that it is very flexible. However, it should be looked on as a temporary solution. It carries the danger that architects will be oversubscribed and constantly distracted by the many simultaneous problems on which they are working; they may be unable to devote appropriate attention to any of the projects.

17.4 HOW DO SOFTWARE ARCHITECTS RELATE TO OTHER STAKEHOLDERS?

It should be clear by now that a software architect exists neither outside nor above his organization. Instead, an architect is an integral part of a software development team, project, and organization. The job of an architect carries with it many responsibilities and

to fulfill those responsibilities, the architect is forced to delegate and rely on co-workers; communicate with and garner support from managers; and keep customers and users in the loop as necessary. We now outline the nature of these relationships, and the likely challenges an architect will encounter in maintaining them.

17.4.1 Architects and Engineers

A large project typically will have a number of engineering teams working on it. Different teams may be charged with different facets of the system, and may include specialists in specific software engineering subdisciplines such as requirements, quality assurance, implementation, and cross-application middleware.

The architect must have a close, well-functioning relationship with all engineers. In the case of some, such as requirements engineers, the relationship may be more natural and objectives more mutually aligned than in the case of others, such as software testers. In either case, however, the guiding concern must be the overall success of the project.

Perhaps the most important constituency within the organization to the architect is the programmers. Programmers will realize and refine the architect's vision. They also serve as validators of the architecture: If an architectural design decision has unforeseen consequences, a programmer likely will be the first one to notice. In such cases especially, a close, cooperative relationship between architects and programmers is critical; if it is lacking, architectural degradation will quickly occur.

In many companies, architects also must work closely with systems engineers and, in some projects such as embedded systems projects, hardware engineers. An architect's relationship with these engineers will likely be different from that with software engineers. One reason is that a software architect creates the overall vision and design for the *software* portion of the given system, but does not have analogous control over the remainder of the system. Another reason is that, because systems and hardware engineers lack training in software engineering, they may not have appropriate appreciation for the nuances of the system's software architectural design. They may fall victim to the misconception that, because software is infinitely malleable in principle, it should always change to accommodate the needs of other system components. By now it should be clear that frequent changes to a system, especially if undertaken hastily, tend to invalidate the key architectural design decisions, and result in architectural degradation.

A related, and even bigger, issue that may occur in the relationship between a software architect and nonsoftware engineers is the lack of respect for the architect's importance to the project. This can, of course, happen with software engineers as well. All engineers working on a given project may be highly capable and may qualify as experts in their own focus areas, whether systems, hardware, or some particular specialty. They may thus feel that they understand the nuances of their task much better than the software architect does. Furthermore, they may not see the direct benefit of strictly adhering to the architecture: They frequently are more interested in achieving local optima due to their limited vision of the entire project. In some development organizations software architecture—and software *architects*—still are not held in very high regard. Coupled with a dislike of being told what to do, this can result in significant resistance to the software architect and contribute to a failed project. Avoiding, or identifying and remedying such situations, is one of the main challenges and primary duties of a software architect.

17.4.2 Architects and Managers

At times, an architect must achieve his objectives in the face of skeptical and even uncooperative engineers, and often must do so without any formal authority over those engineers. The architect may have to resort to winning them over with personal competence and leadership. Clearly, things are likely to go more smoothly if he also has the support of the organization's management.

Software development organizations are increasingly beginning to understand the importance of software architectures and software architects. Management must work to ensure a project's success and, in an ideal world, that would mean ensuring that the system's architecture, that is, the architects' vision, is realized fully and effectively. In turn, this requires architects and managers to interact frequently and to closely align their goals. This may mean that the managers must adjust their expectations of a project based on the architects' technical considerations, and it may also require architects to adjust their expectations, and the architecture itself, based on the organization's realities. Schedule and budget constraints, marketplace considerations, personnel turnover, and other such issues must be considered.

Managers and architects do not always work well together, however. There may be many reasons for this, but from a software architect's perspective one reason can be particularly damaging: Understanding the importance of the architect's role in a project does not ensure managerial competence. A project, and an entire organization, may be mismanaged in many ways and for many reasons. The managers also may simply lack the technical know-how to grasp the importance of the architect's design decisions. As a result, while a manager may in principle support the *architect*, he may unwittingly undermine the *architecture*.

Managers may fail to think through all of the consequences of their actions, which may negatively impact the system's architecture. They may commit to purchasing and using third-party components, frameworks, or middleware that are a poor fit for the architecture. They may decide to comply with standards that cannot be easily shoe-horned into the company's existing architectures. They also may commit to delivering too much, too quickly, too cheaply, without understanding the architectural implications of those promises. An architect must always be aware and may be forced to respond to these challenges.

Of course, an even bigger challenge faced by architects is in more traditional software development shops, in which certain techniques, tools, and processes may have ossified over time. In some parts of such organizations, architecture-centric development on the whole may be considered to be a fad, too inefficient, and unnecessary. Likewise, a particular architectural approach—for example, requiring that a developer request a formal architectural design review before he may change a component's interface—may go against the organization's culture, which in the past has relied on, and thus still encourages, a swaggering form of individual initiative. In such situations, architects simultaneously must fight two battles: arriving at an effective architecture for the system under construction while at the same time trying to overcome institutional resistance, ignorance, and hubris.

17.4.3 Other Stakeholders

Software architects must also interact with additional stakeholders, both within and outside of their organization. Of particular importance is their relationship with the

marketing department. In many organizations, innovation is frequently driven by market needs and pressures. This is, of course, understandable because a company's primary goal is to sell products. There is nothing inherently wrong or inverted in such an arrangement: The marketing department researches the current marketplace conditions and customers' needs, and reports those to the company's management or engineers; in turn, the software architects employ their skills to design systems that will best take advantage of the market conditions and satisfy customer needs. It is, therefore, not unusual for marketing to drive a product's development, and hence its architecture: It may determine the systems that need to be built, their exact features, development priorities, release dates, and so forth.

At the same time, there are certain pitfalls in this relationship. Both the architects and marketing staff may fail to properly understand the motivations, abilities, and expertise of the other. The architects may resist being told what to do, especially if they fail to appreciate that market conditions will occasionally get in the way of producing the best or preserving the originally envisaged architecture for a system. For example, an architect's task will be directly impacted if a potential competitor has been identified and it becomes imperative to release a product much earlier than originally planned. So, too, if the primary customer base for the system under construction is observed to be moving to a different computing platform, or if the envisioned customer base significantly expands or shrinks. Architects have no choice but to accommodate such nontechnical realities.

On the other hand, the marketing department may cause problems if it fails to understand that a system's architecture cannot be arbitrarily or abruptly modified to incorporate newly identified needs. Marketing may also strive for unrealistic goals and unreasonable deadlines, and may attempt to drive the engineering process without adequate understanding of technical challenges and realities. In the most extreme case, marketing may promise a product that simply cannot be built by the company's engineers at the given time.

In either case, misunderstandings and lack of proper communication between architecture and marketing teams may result in an adversarial, counterproductive relationship. Keeping this relationship healthy should be an important, regular objective.

Somewhat similar to the architect's relationship with the marketing staff will be his relationship with the system's customers and users, with whom the architect may need to interface directly. In fact, he may well be the face of the project as far as the customers and users are concerned: The architect may present the project vision to them, ensure that their requirements are properly reflected in the system's blueprint and eventually in the system itself, address their inquiries, and resolve their concerns and doubts.

If properly established and nurtured, this can be a very productive relationship, helping ensure the system's final acceptance, with all of the major issues addressed as they came up during construction. On the other hand, the customers and users may be very particular in their requests, impatient, and, in the worst case, insufficiently knowledgeable of the nature of software and the nuances of its development. Thus, in addition to providing the rationale behind the architecture to the stakeholders within the organization (developers, testers, managers), the architect may also need to tailor that rationale for the benefit of the technically less savvy customers and users.

17.5 REMAINING CHALLENGES

The architect must possess many skills, from maintaining the system's conceptual integrity to serving as the project's cheerleader, leading by the strength of technical knowledge and character, and interacting effectively with a broad range of stakeholders with frequently competing goals. While some of these skills cannot be taught, they certainly can be nurtured. At the same time, there are several other facets of an architect's job for which he can be trained in a conventional manner—an observation that was a motivation for this textbook as well.

The principal challenge for software development organizations—including, but not restricted to, the traditional software shops that want to introduce software architects and explicit architecture-based software development for the first time—is identifying the best candidates for the software architect positions, the appropriate skill sets they should possess, and the people who can train them. There are several examples in industry of such training programs. One example is NASA Jet Propulsion Laboratory's Software Architect Program (Vickers 2004). Such programs are relatively recent, and their effectiveness and required adjustments will need to be monitored over time.

17.6 END MATTER

A software architect has a multifaceted job, teeming with unique challenges and substantial responsibility. An architect may work alone or, more likely, as part of a team. His job description may be somewhat imprecise, but it will involve many technical skills discussed in this book: architecture modeling and analysis; awareness of and proficiency with architectural styles and patterns; familiarity with product lines and domain-specific architectures; understanding of relevant implementation, deployment, and evolution technologies; and so on. The job of a software architect also requires many skills that, at first blush, may not seem necessary for a software practitioner or that may not be readily learned from a book, including the ability to communicate effectively with many different types of stakeholder; to ensure compliance with relevant laws, regulations, and standards; to lead; to ensure the architect's economic viability. As an architect, one thus truly has to be a jack-of-all-trades. This is yet another facet of our discipline that distinguishes it from other engineering endeavors.

A critical success factor for a software system is that system's architecture. The skills of the architects and composition of the architecture teams therefore will have a direct and significant impact on the success of a project and of an organization. Each individual architect must possess a large number of diverse skills, while the architecture teams must have a comprehensive balance of those skills.

Fred Brooks has said that great designers must be spotted, selected, and nurtured. However, as we have seen, an architect's job is not restricted to design. He must possess a broad cross-section of skills. Many of those skills can be learned; all can be improved with study and practice. It is thus in a company's interest to provide the appropriate training for its architects. It is also important to strive to achieve a careful balance in the architecture teams.

The bottom line is that a company with skilled architects and carefully composed architecture teams will have a better opportunity to succeed in the marketplace.

The Business Case

17.7 REVIEW QUESTIONS

1. What are the necessary skills a software architect should possess?
2. What are some additional useful skills?
3. This chapter discussed several bad habits a software architect may exhibit. Can you identify any others? Be sure to justify your answers.
4. Your organization has identified an exceptional young software designer and would like to promote him into a software architect. What, if any, additional training would be required to accomplish this? How would you suggest that your organization proceed?
5. Based on the discussion provided in this chapter, present an argument why purchasing an architecture can be a recipe for failure.
6. Outsourcing software development has proven to be a successful strategy for many organizations. Should organizations also outsource architectural design? Be sure to carefully state your argument.

17.8 FURTHER READING

Brooks was one of the first to recognize the importance of a chief designer in a software project in his seminal book, *The Mythical Man-Month* (Brooks 1995). Many of his observations, drawn from the experience of building the IBM 360 operating system in the 1960s, still ring true several decades later and inspired several of the points in this chapter. Brooks followed that with an essay arguing for the importance of software design and of great software designers in tackling the essential difficulties of software engineering (Brooks 1987). The dangers of the Peter Principle—which, as this chapter has argued, has a direct relevance to software organizations and architects—are described in Peter's book of the same name (Peter and Hull 1969).

More recently, a number of authors have tried to answer the questions of who software architects are, what skills they must possess, and how they go about devising a system's architecture. Several of these issues were considered, at least indirectly, by Curtis, Krasner, and Iscoe (Curtis, Krasner, and Iscoe 1988) in the context of designing large-scale software systems, and then more directly by Kruchten (Kruchten 1995, 1999). Kruchten's 1999 paper, in particular, provides an excellent elaboration of the

desired skills and responsibilities of software architects. A number of insights from that paper have found their way into this chapter. Several similar observations have also been made by other authors, such as Kari Smolander (Smolander 2002) and Tero Päiväranta (Smolander and Päiväranta 2002).

Elisabetta DiNitto and David Rosenblum (Di Nitto and Rosenblum 1999) provide one of many examples of the manner and extent to which technological solutions (in their case, middleware) impact a software system's architecture, and thereby the system's architects. Will Tracz's work on domain-specific software architectures (Tracz, 1994; Tracz, 1995; Taylor, Tracz, and Coglianese 1995) argues for the other critical facet of an architect's job—domain expertise.

In practice, as argued in this chapter, an effective software architect must possess many additional skills. A number of organizations have established software architect training programs with the objective of molding their architects into well rounded, multifaceted employees. One representative example is NASA Jet Propulsion Laboratory's Software Architect Program (Vickers 2004).

Bibliography

- Abbott, Russell J. 1983. Program Design by Informal English Descriptions. *Communications of the ACM* 26 (11): 882–894.
- Abdul-Rahman, Alfarez, and Stephen Hailes. 2000. Supporting Trust in Virtual Communities. In Hawaii International Conference on System Sciences, Jan. 4–7, (Maui, HI).
- Albin, Stephen T. 2003. *The Art of Software Architecture: Design Methods and Techniques*, Wiley Publishing, Inc.
- Alexander, C. 1979. *The Timeless Way of Building*. New York, Oxford University Press.
- Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, USA.
- Allen, Robert. 1997. A Formal Approach to Software Architecture. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh.
- Allen, Robert, and David Garlan. 1997. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6 (3): 213–249.
- Apache Software Foundation. *Apache Derby* (Website). Available from db.apache.org/derby/.
- Apollo 11: *The NASA Mission Reports*. 1999. Edited by R. Godwin, Apogee Books Space Series, Collector's Guide Publishing, Inc.
- Argo/UML. 2000. Available from argouml.tigris.org/.
- Atria Software. 1992. ClearCase Concepts Manual.
- Balek, D., and F. Plasil. 2001. Software Connectors and Their Role in Component Deployment. In DAIS '01, Krakow, Kluwer.
- Baset, Salman A., and Henning Schulzrinne. 2004. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol, Columbia University, CUCS-039-04.
- Bastarrica, M., A. Shvartsman, and S. Demurjian. 1998. A Binary Integer Programming Model for Optimal Object Distribution. In Second International Conference on Principles of Distributed Systems, December 1998, (Amiens, France), Hermes, pp. 211–226.
- Batory, Don, Lou Coglianese, Steve Shafer, and Will Tracz. 1995. The ADAGE Avionics Reference Architecture. In AIAA Computing in Aerospace-10 Conference, March.
- Batory, Don, David McAllester, Lou Coglianese, and Will Tracz. 1995. Domain Modeling in Engineering Computer-Based Systems. In 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, March, pp. 19–26.
- Berners-Lee, Tim, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. 1994. The World-Wide Web. *Communications of the ACM* 37 (8): 76–82.
- Binns, Pam, Matt Englehart, Mike Jackson, and Steve Vestal. 1996. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering* 6 (2): 201–227.
- Bishop, Matt. 2003. *Computer Security: Art and Science*, Addison-Wesley.

- Boehm, B.W. 1988. A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21 (5): 61–72.
- Boehm, Barry W. 1981. *Software Engineering Economics*. Upper Saddle River, NJ, Prentice Hall PTR.
- Booch, Grady 1986. Object-Oriented Development. *IEEE Transactions on Software Engineering* 12 (2): 211–221.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1998. *The Unified Modeling Language User Guide*, Object Technology Series. Reading, MA, Addison-Wesley Professional.
- . 2005. *The Unified Modeling Language User Guide*. 2nd ed., Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading, MA.
- Bosch, Jan. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Reading, MA, ACM Press, Addison-Wesley Professional.
- Bowman, Ivan T., Richard C. Holt, and Neil V. Brewster. 1999. Linux as a Case Study: Its Extracted Software Architecture. In *Twenty-first International Conference on Software Engineering (ICSE'99)*, May 16–22, (Los Angeles, CA), pp. 555–563.
- Brand, Stewart 1994. *How Buildings Learn: What Happens After They're Built*, Penguin Books.
- Bray, Tim, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML): Part I. Syntax (Recommendation)*. World Wide Web Consortium, February, 1998. Available from www.w3.org/TR/1998/REC-xml.
- Briand, Lionel C., and Alexander L. Wolf, eds. 2007. *Future of Software Engineering (FoSE 2007)*. Minneapolis, MN: IEEE Computer Society.
- Brooks, Frederick P. Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering*. 2nd ed., Addison-Wesley.
- . 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* 20 (4): 10–19.
- Brooks, Rodney A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* 2 (1): 14–23.
- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, West Sussex, UK, Wiley.
- Bush, Vannevar. 1996. As We May Think. *interactions* 3 (2): 35–46.
- Carzaniga, A., A. Fuggetta, André van der Hoek, R. S. Hall, D. M. Heimbigner, and A. L. Wolf 1998. A Characterization Framework for Software Deployment Technologies, University of Colorado, Boulder, CU-CS-857-98.
- Carzaniga, Antonio, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems* 9 (3): 332–383.
- The Catalog of Compiler Construction Tools*. 2006. Fraunhofer Institute for Computer Architecture and Software Technology. Available from catalog.compilertools.net/.
- Chapman, Merrill R. (Rick). 2003. In *Search of Stupidity: Over 20 Years of High-Tech Marketing Disasters*. Apress.
- Clements, P., F. Bachmann, L. Bass, David Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. 2002. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.
- Clements, Paul C., and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns*, The SEI Series in Software Engineering, Addison-Wesley Professional.
- Clements, Paul, Rick Kazman, and Mark Klein. 2002. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional.

- Coglianesi, Lou, Roy Smith, and Will Tracz. 1992. DSSA Case Study: Navigation, Guidance, and Flight Director Design and Development. In *IEEE Symposium on Computer-Aided Control System Design*, March, pp. 102–109.
- Cohen, Bram. *Incentives Build Robustness in BitTorrent*. BitTorrent.org, May 22, 2003. Available from www.bittorrent.org/bittorrentecon.pdf.
- Collins-Sussman, Ben, Brian W. Fitzpatrick, and C. Michael Pilato. 2004. *Version Control with Subversion* (HTML), 2004. Available from svnbook.red-bean.com/.
- Colouris, G., J. Dollimore, and T. Kindberg. 1994. *Distributed Systems: Concepts and Design*. 2nd. Ed., Addison-Wesley.
- Cook, J., and A. Orso. 2005. MonDe: Safe Updating through Monitored Deployment of New Component Versions. In *Sixth ESEC/FSE Workshop on Program Analysis for Software Tools and Engineering*, September, (Lisbon, Portugal).
- Coram, Robert. 2002. *Boyd: The Fighter Pilot Who Changed the Art of War*, Little, Brown, and Company.
- Curtis, B., H. Krasner, and N. Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* 31 (11): 1268–1287.
- Damiani, Ernesto, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. 2002. A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks. In *Ninth ACM Conference on Computer and Communications Security*, November (Washington, DC).
- Dashofy, E. M. 2001. Issues in Generating Data Bindings for an XML Schema-Based Language. In *Workshop on XML Technologies in Software Engineering (XSE 2001)*, May 15, (Toronto, Canada).
- . 2003. *xADL 2.0 Distilled: A Guide for Users of the xADL 2.0 Language*. (Webpage), January 2003. Available from www.isr.uci.edu/projects/xarchuci/guide.html.
- . 2007. Supporting Stakeholder-Driven, Multi-View Software Architecture Modeling, Ph.D. dissertation, Information and Computer Science, University of California–Irvine, Irvine, CA.
- Dashofy, E. M., N. Medvidovic, and R. N. Taylor. 1999. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Twenty-First International Conference on Software Engineering (ICSE '99)*, May 16–22, (Los Angeles, CA), pp. 3–12.
- Dashofy, Eric M., and André van der Hoek. 2001. Representing Product Family Architectures in an Extensible Architecture Description Language. In *International Workshop on Product Family Engineering (PFE-4)*, October, pp. 330–341.
- Dashofy, Eric, André van der Hoek, and Richard N. Taylor. 2005. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14 (2): 199–245.
- Dean, Jeffrey, and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December (San Francisco, CA).
- DeLine, R. 2001. Avoiding Packaging Mismatch with Flexible Packaging. *IEEE Transactions on Software Engineering* 27 (2): 124–143.
- DeRemer, Frank, and Hans H. Kron. 1976. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2 (2): 80–86.
- Deutsch, Peter, and James Gosling. *The Eight Fallacies of Distributed Computing*, 1994. Available from blogs.sun.com/jag/resource/Fallacies.html.

- Devanbu, Premkumar T., and Stuart Stubblebine. 2000. Software Engineering for Security: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, (Limerick, Ireland), ACM Press, pp. 227–239.
- Di Nitto, Elisabetta, and David S. Rosenblum. 1999. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Twenty-First International Conference on Software Engineering*, May, (Los Angeles, CA), IEEE Computer Society, pp. 13–22.
- Dijkstra, Edsger W. 1968. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM* 11 (3): 147–148.
- Dilley, J., B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. 2002. Globally Distributed Content Delivery. *IEEE Internet Computing* 6 (5): 50–58.
- Dillinger, Marcus, Kambiz Madani, and Nancy Alonistioti. 2003. *Software Defined Radio: Architectures, Systems and Functions*, Wiley.
- Dillon, Laura A., George Kutty, P. Michael Melliar-Smith, Louise E. Moser, and Y.S. Ramakrishna. 1992. Graphical Specifications for Concurrent Software Systems. In *Fourteenth International Conference on Software Engineering*, May, (Melbourne, Australia), pp. 214–224.
- Dinda, Peter, Thomas Gross, Roger Karrer, and Bruce Lowekamp. 2001. The Architecture of the Remos System. In *IEEE International Symposium on High-Performance Distributed Computing (HPDC 2001)*, August, (San Francisco, CA).
- Dingledine, Roger, Michael Freedman, David Molnar, David Parkes, and Paul Syverson. 2003. Reputation. In *The Digital Government Civic Scenario Workshop*, April 28–29, (Harvard University, Cambridge, MA, USA).
- Dobrica, L., and E. Niemela. 2002. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering* 28 (7): 638–53.
- DoD Architecture Framework Working Group. 2004. *DoD Architecture Framework*, Version 1.0, United States Department of Defense.
- Doherty, P., P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, and B. Wingman. 2004. A Distributed Architecture for Autonomous Unmanned Aerial Vehicle Experimentation. In *Seventh International Symposium on Distributed Autonomous Systems*.
- Dwyer, M., J. Hatcliff, C. Pasareanu, Robby, and W. Visser. 2007. Formal Software Analysis: Emerging Trends in Software Model Checking. In *Future of Software Engineering 2007*, edited by L. Briand and A. Wolf: IEEE-CS Press, pp. 120–136.
- Edwards, George, Sam Malek, and Nenad Medvidović. 2007. Scenario-Driven Dynamic Analysis of Distributed Architectures. In *Proceedings of the Tenth International Conference on Fundamental Approaches to Software Engineering (FASE)*, (Braga, Portugal), Springer: Lecture Notes in Computer Science 4422, pp. 125–139.
- Emmerich, Wolfgang. 2000. *Engineering Distributed Objects*, Wiley.
- Eyles, Don. 2004. Tales from the Lunar Module Guidance Computer. In *Twenty-Seventh Annual Guidance and Control Conference of the American Astronautical Society (AAS)*, February 6, (Breckenridge, CO).
- Farrache, Gilles. *bbFTP* (Website), 2005. Available from doc.in2p3.fr/bbftp/.
- Feijs, Loe, and Roel De Jong. 1998. 3D Visualization of Software Architectures. *Communications of the ACM* 41 (12): 73–78.
- Feiler, Peter H., Bruce Lewis, and Stephen Vestal. 2003. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*. May (Washington, DC).

- Fielding, R., J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. Hypertext Transfer Protocol—HTTP/1.1, Internet Engineering Task Force, Request for Comments, RFC 2616.
- Fielding, Roy T., and Richard N. Taylor. 2002. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)* 2 (2): 115–150.
- Fielding, Roy Thomas. 2000. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. dissertation, Information and Computer Science, University of California—Irvine, Irvine, CA.
- Finkelstein, Anthony. 2000. *The Future of Software Engineering 2000*. New York, ACM Press.
- Firby, Robert James. 1989. Adaptive Execution in Complex Dynamic Worlds. Doctoral thesis, Computer Science, Yale University.
- Fogel, K. 1999. *Open Source Development with CVS*. Scottsdale, Coriolis.
- Foote, Brian, and Joseph Yoder. 1997. Big Ball of Mud. In Fourth Conference on Patterns Languages of Programs (PLOP97/EuroPLOP97), (Monticello, IL), Technical Report #WUCS-97-34, Department of Computer Science, Washington University, p. 21.
- Formal Systems (Europe) Ltd. 2005. *Failures-Divergence Refinement: FDR2 Users Manual*.
- Foster, Ian, Carl Kesselman, and Steven Tuecke. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications* 15 (3): 200–222.
- Fuggetta, Alfonso, Gian Pietro Picco, and Giovanni Vigna. 1998. Understanding Code Mobility. *IEEE Transactions on Software Engineering* 24 (5): 342–361.
- Gambetta, Diego. 2000. Can We Trust Trust? In *Trust: Making and Breaking Cooperative Relations*, edited by D. Gambetta. Oxford: Department of Sociology, University of Oxford, pp. 213–237.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series. Reading, MA, Addison-Wesley Professional.
- Garlan, D., R. Allen, and J. Ockerbloom. 1995. Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts. In Seventeenth International Conference on Software Engineering (ICSE), ACM, pp. 179–185.
- . 1994. Exploiting Style in Architectural Design Environments. In Second ACM SIGSOFT '94 Second Symposium on the Foundations of Software Engineering, December 6–9, (New Orleans, LA), ACM Press, pp. 175–188.
- . 1995. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software* 12 (6): 17–26.
- Garlan, David, S. Cheng, and Bradley Schmerl. 2003. Increasing System Dependability through Architecture-Based Self-Repair. In *Architecting Dependable Systems*, edited by R. de Lemos, C. Gacek and A. Romanovsky.
- Garlan, David, Robert T. Monroe, and David Wile. 1997. ACME: An Architecture Description Interchange Language. In CASCON '97, November, (Toronto, Ontario, Canada), IBM Center for Advanced Studies, pp. 169–183.
- . 2000. ACME: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, edited by G. T. Leavens and M. Sitaraman: Cambridge University Press, pp. 47–68.
- Gasser, Morrie. 1988. *Building a Secure Computer System*, Van Nostrand Reinhold New York Co.

- Georgas, John C., and Richard N. Taylor. 2004. Towards a Knowledge-based Approach to Architectural Adaptation Management. In First ACM SIGSOFT Workshop on Self-Managed Systems (WOSS '04), October (Newport Beach, CA), ACM Press, pp. 59–63.
- Georgas, John C., André van der Hoek, and Richard N. Taylor. 2005. Architectural Runtime Configuration Management in Support of Dependable Self-Adaptive Software. In Workshop on Architecting Dependable Systems (WADS '05), in conjunction with ICSE 2005 (St. Louis, MO), pp. 48–53.
- Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, (Bolton Landing, NY), ACM, pp. 29–43.
- Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli. 2003. *Fundamentals of Software Engineering*. 2nd ed., Upper Saddle River, NJ, Prentice Hall.
- Gokhale, Swapna. 2007. Architecture-Based Software Reliability Analysis: Overview and Limitations. *IEEE Transactions on Dependable and Secure Computing* 4 (1): 32–40.
- Goldman, N., and R. Balzer. 1999. The ISI Visual Design Editor Generator. In IEEE Symposium on Visual Languages (Tokyo), pp. 20–27.
- Gomaa, Hassan. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley Professional.
- Gomaa, Hassan, and Mohamed Hussein. 2004. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. In Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, pp. 79–88.
- Gorlick, Michael M., and Rami R. Razouk. 1991. Using Weaves for Software Construction and Analysis. In Thirteenth International Conference on Software Engineering, May, pp. 23–34.
- Gorton, Ian. 2008. Software Architecture Challenges for Data Intensive Computing. In Seventh Working IEEE/IFIP Conference on Software Architecture, February, (Vancouver, BC), IEEE Computer Society, pp. 4–6.
- Gorton, Ian, Adam Wynne, Justin Almquist, and Jack Chatterton. 2008. The MeDiCi Integration Framework: A Platform for High Performance Data Streaming Applications. In Seventh Working IEEE/IFIP Conference on Software Architecture, (Vancouver, BC), IEEE Computer Society, pp. 95–104.
- Gröne, Bernhard, Andreas Knöpfel, and Rudolf Kugel. 2002. Architecture Recovery of Apache 1.3—A case study. In 2002 International Conference on Software Engineering Research and Practice, (Las Vegas).
- Gröne, Bernhard, Andreas Knöpfel, Rudolf Kugel, and Oliver Schmidt. 2004. The Apache Modeling Project, Hasso Plattner Institute for Software Systems Engineering.
- Grosso, William. 2001. *Java RMI*. 1st ed., O'Reilly Media, Inc.
- Gutttag, John V., James J. Horning, and Jeannette M. Wing. 1985. The Larch Family of Specification Languages. *IEEE Software* 2 (5): 24–36.
- Guttman, Barbara, and Edward Roback. 1995. An Introduction to Computer Security: The NIST Handbook, edited by Computer Systems Laboratory: National Institute of Standards and Technology: U.S. Government Printing Office.
- Haas, R., P. Droz, and B. Stiller. 2003. Autonomic Service Deployment in Networks. *IBM Systems Journal* 42 (1): 150–164.
- Hall, Richard S., Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf. 1997. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In

- 1997 International Conference on Distributed Computing Systems, May, (Baltimore, MD), IEEE Computer Society, pp. 269–278.
- Hall, Richard S., Dennis Heimbigner, and Alexander L. Wolf. 1999. A Cooperative Approach to Support Software Deployment Using the Software Dock. In 1999 International Conference on Software Engineering, May, (Los Angeles, CA), ACM Press, pp. 174–183.
- Harel, David. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8: 231–274.
- Harold, Elliott Rusty. 2006. *Java I/O*. 2nd ed., O'Reilly Media, Inc.
- Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman. 1976. Protection in Operating Systems. *Communications of the ACM* 19 (8): 461–471.
- Hayes-Roth, B., K. Pfleger, P. Lalande, P. Morignot, and M. Balabanovic. 1995. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. *IEEE Transactions on Software Engineering* 21 (4): 288–301.
- Heimdahl, Mats P. E. 2007. Safety and Software Intensive Systems: Challenges Old and New. In *Future of Software Engineering 2007*, edited by L. Briand and A. Wolf: IEEE-CS Press, pp. 137–152.
- Heineman, George T., and William T. Councill, eds. 2001. *Component-Based Software Engineering: Putting the Pieces Together*. Reading, MA: Addison-Wesley.
- Hendrickson, Scott A., Eric M. Dashofy, and Richard N. Taylor. 2005. An (Architecture-Centric) Approach for Tracing, Organizing, and Understanding Events in Event-Based Software Architectures. In Thirteenth International Workshop on Program Comprehension, in conjunction with ICSE 2005, May 15–16, (St. Louis, MO), pp. 227–236.
- Hendrickson, Scott A., and André van der Hoek. 2007. Modeling Product Line Architectures through Change Sets and Relationships. In Twenty-Ninth International Conference on Software Engineering (ICSE 2007), May 20–26, (Minneapolis, MN).
- Heng, Christopher. *Free Compiler Construction Tools* (Website), October 9, 2007. Available from www.thefreecountry.com/programming/compilerconstruction.shtml.
- Hirsch, D., S. Uchitel, and D. Yankelevich. 1999. Towards a Periodic Table of Connectors. In *Seminario en Tecnología de Software*, (Bueno Aires, Argentina).
- Hitchens, Ron. 2002. *Java NIO*, O'Reilly Media, Inc.
- Hoare, C. A. R. 1978. Communicating Sequential Processes. *Communications of the ACM* 21 (8): 666–677.
- Hofmeister, Christine, R. L. Nord, and D. Soni. 1999. *Applied Software Architecture*, Addison Wesley Longman.
- Honeywell, Inc. *MetaH Evaluation and Support Site*, 1998. Available from www.htc.honeywell.com/metah/.
- Houston, Peter 1998. Building Distributed Applications with Message Queuing Middleware, Microsoft Corporation, White Paper.
- Hunt, Galen C., and Michael L. Scott. 1999. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February (New Orleans, LA), USENIX.
- IBM. *Autonomic Computing Manifesto* (PDF document), 2001. Available from www.research.ibm.com/autonomic/manifesto/autonomic-computing.pdf.
- . 2002. *Autonomic Computing Architecture: A Blueprint for Managing Complex Computing Environments*, Whitepaper.

- . 2003. *WebSphere MQ - Product Overview* (Website). IBM. 2003. Available from www-3.ibm.com/software/ts/mqseries/messaging/.
- IEEE-SA Standards Board. 2000. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. New York, NY, The Institute of Electrical and Electronics Engineers, Inc.
- IEEE. 1991. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*: 610. New York: Institute of Electrical and Electronics Engineers, pp. 217.
- . 2004. IEEE Std 1003.1, 2004 Edition.
- Immonen, Anne, and Eila Niemela. 2007. Survey of Reliability and Availability Prediction Methods from the Viewpoint of Software Architecture. *Springer Journal of Software and System Modeling*.
- InstallShield Corporation. 2000. *InstallShield* (HTML). 2000. Available from www.installshield.com/.
- ISO. 1989. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior. (ISO 8807).
- ISO/IEC 1998. Information Technology—Open Distributed Processing—Reference Model, ISO/IEC, 10746.
- Jackson, Daniel. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2): 256–290.
- Jackson, M. 2001. *Problem Frames*. Reading, MA, Addison-Wesley Professional.
- Jackson, Michael. 1995. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, ACM Press/Addison-Wesley.
- . 2000. Problem Analysis and Structure. In *Engineering Theories of Software Construction*. NATO Summer School., August, (Marktobderdorf, Germany), IOS Press, pp. 3–20.
- . 2007. *Michael Jackson—Consultancy & Research in Software Development. Past Research Topics*. (Website), 2007. Available from mcs.open.ac.uk/mj665/topics.html.
- Jacobson, Ivar. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. 1st ed., Addison-Wesley Professional.
- JavaSoft 1996. *JavaBeans 1.0 API Specification*, Sun Microsystems, Inc., version 1.00-A.
- . 2001. *Java Message Service API*, Sun Microsystems, Inc., Version 1.0.2b.
- Jones, J. Christopher. 1970. *Design Methods: Seeds of Human Futures*. New York, John Wiley & Sons, Ltd.
- . 1992. *Design Methods*. 2nd ed. New York, NY, John Wiley & Sons, Inc. Original edition, J. Christopher Jones, *Design Methods: Seeds of Human Futures*.
- Kan, Gene. 2001. Gnutella. In *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, edited by A. Oram: O'Reilly, pp. 94–122.
- Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, Technical, CMU/SEI-90-TR-21.
- Kelley, Tom, Jonathan Littman, and Tom Peters. 2001. *The Art of Innovation: Lessons in Creativity from IDEO, America's Leading Design Firm*. New York, Currency/Doubleday.
- Kephart, Jeffrey O., and David M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36 (1): 41–50.
- Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed., Prentice Hall.

- Khare, Rohit, and Richard N. Taylor. 2004. Extending the REpresentational State Transfer Architectural Style for Decentralized Systems. In Twenty-Sixth International Conference on Software Engineering (ICSE 2004), May (Edinburgh, Scotland, UK), pp. 428–437.
- Kichkaylo, Tatiana, Anca Ivan, and Vijay Karamcheti. 2003. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. In International Parallel and Distributed Processing Symposium (IPDPS'03).
- Klusener, A. S., L. R. Lämmel, and C. Verhoef. 2005. *Architectural Modifications to Deployed Software*. Vol. 54, Elsevier North-Holland, Inc.
- Kramer, J., and J. Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering* 16 (11): 1293–1306.
- Kramer, Jeff, and Jeff Magee. 1988. Change Management of Distributed Systems. In Proceedings of the Third Workshop on ACM SIGOPS European Workshop: Autonomy or Interdependence in Distributed Systems? (Cambridge, United Kingdom), ACM Press, pp. 1–4.
- Kruchten, Philippe. 1995. Mommy: Where Do Software Architectures Come From? In First International Workshop on Architectures for Software Systems, April, (Seattle, WA).
- . 1999. The Software Architect—and the Software Architecture Team. In Software Architecture. Proceedings of TC2 First Working IFIP Conference on Software Architecture, (San Antonio, TX), Kluwer Academic Publishers, pp. 565–583.
- . 2000. *The Rational Unified Process: An Introduction*. Reading, MA: Addison-Wesley Professional.
- Kruchten, Phillippe. 2003. *The Rational Unified Process: An Introduction*. 3rd ed.
- Lampson, Butler W. 1974. Protection. *ACM SIGOPS Operating Systems Review* 8 (1): 18–24.
- Larman, Craig. 2002. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2nd ed, Prentice-Hall PTR.
- Lazowska, Edward D., John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Upper Saddle River, NJ, Prentice-Hall, Inc.
- Lee, Seungjoon, Rob Sherwood, and Bobby Bhattacharjee. 2003. Cooperative Peer Groups in NICE. In IEEE Infocom, April 1–3, (San Francisco).
- Leveson, Nancy G. 1995. *Safeware: System Safety and Computers*, Addison-Wesley Professional.
- Liang, Sheng. 1999. *Java Native Interface: Programmer's Guide and Specification*. 1st ed., Prentice Hall PTR.
- Littlewood, Bev, and Lorenzo Strigini. 2000. Software Reliability and Dependability: A Roadmap. In *The Future of Software Engineering 2000*, (Limerick, Ireland), ACM Press, pp. 175–188.
- Liu, C. L., and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)* 20 (1): 46–61.
- Luckham, David. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley.
- Luckham, David C., and Friedrich W. von Henke. 1985. An Overview of Anna, a Specification Language for Ada. *IEEE Software* 2 (2): 9–23.
- Luckham, David C., John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. 1995. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering* 21 (4): 336–355.
- Luckham, David C., and James Vera. 1995. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering* 21 (9): 717–734.

- Lutz, Robyn. 2000. Software Engineering for Safety: A Roadmap. In *The Future of Software Engineering 2000*, (Limerick, Ireland), ACM Press, pp. 213–224.
- Lyu, Michael. 2007. Software Reliability Engineering: A Roadmap. In *Future of Software Engineering 2007*, edited by L. Briand and A. Wolf: IEEE-CS Press, pp. 153–170.
- Magee, Jeff, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. 1995. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference (ESEC 95)*, Springer-Verlag, Berlin, pp. 137–153.
- Magee, Jeff, Naranker Dulay, and Jeff Kramer. 1994. Regis: A Constructive Development Environment for Distributed Programs. *IEEE/IOP/BCS Distributed Systems Engineering* 1 (5): 304–312.
- Magee, Jeff, and Jeff Kramer. 1996. Dynamic Structure in Software Architectures. In *Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, October 16–18, (San Francisco), ACM SIGSOFT, pp. 3–14.
- . 2006. *Concurrency: State Models and Java Programs*. 2nd ed., Wiley.
- Maier, Mark, and Eberhardt Rechtin. 2000. *The Art of Systems Architecting*. 2nd ed., Boca Raton, FL, CRC Press.
- Malek, Sam, Nels Beckman, Marija Mikic-Rakic, and Nenad Medvidović. 2005. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In *Architecting Dependable Systems III*, edited by R. de Lemos, C. Gacek and A. Romanowski: Springer Verlag.
- Malek, Sam, Marija Mikic-Rakic, and Nenad Medvidović. 2005. A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Transactions on Software Engineering* 31 (3): 256–272.
- Malek, Sam, Chiyoung Seo, Sharmila Ravula, Brad Petrus, and Nenad Medvidović. 2007. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In *Proceedings of the Twenty-Ninth International Conference on Software Engineering*, IEEE Computer Society, pp. 591–601.
- Manolescu, Ioana, Marco Brambilla, Stefano Ceri, Sara Comai, and Piero Fraternali. 2005. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Transactions on Internet Technology (TOIT)* 5 (3): 439–479.
- Marsh, Stephen. 1994. Formalising Trust as a Computational Concept, Department of Mathematics and Computer Science, University of Stirling, Stirling.
- Mattmann, Chris A., Daniel J. Crichton, Nenad Medvidović, and Steve Hughes. 2006. A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications. In *Proceeding of the Twenty-Eighth International Conference on Software Engineering*, (Shanghai, China), ACM Press, pp. 721–730.
- Mattmann, Chris A., Nenad Medvidović, Paul M. Ramirez, and Vladimir Jakobac. 2005. Unlocking the Grid. In *Component-Based Software Engineering: 8th International Symposium, CBSE 2005*, May 14–15, (St. Louis, MO), Lecture Notes in Computer Science.
- Mattmann, Chris, Sam Malek, N. Beckman, M. Mikic-Rakic, Nenad Medvidović, and Dan Crichton. 2005. GLIDE: A Grid-Based, Lightweight, Infrastructure for Data-Intensive Environments. In *European Grid Conference (EGC2005)*, February 14–16, (Amsterdam, The Netherlands), pp. 68–77.
- Maybee, Mark J., Dennis M. Heimigner, and Leon J. Osterweil. 1996. Multilanguage Interoperability in Distributed Systems. In *Eighteenth International Conference on Software Engineering*, March (Berlin, Germany), IEEE Computer Society, pp. 451–463.
- Medvidovic, Nenad, Eric Dashofy, and Richard N. Taylor. 2007. Moving Architectural Description from Under the Technology Lamppost. *Information and Software Technology* 49 (1): 12–31.

- Medvidović, Nenad, Peyman Oreizy, and Richard N. Taylor. 1997. Reuse of Off-the-Shelf Components in C2-Style Architectures. In Nineteenth International Conference on Software Engineering, May (Boston, MA), ACM Press, pp. 692–700.
- Medvidović, Nenad, and Richard N. Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26 (1): 70–93.
- Mehta, Nikunj R., Nenad Medvidović, and Sandeep Phadke. 2000. Towards a Taxonomy of Software Connectors. In 2000 International Conference on Software Engineering, June 4–11 (Limerick, Ireland), ACM Press, pp. 178–187.
- Microsoft Corporation. 2007. *PowerPoint Home Page—Microsoft Office Online*, 2007. Available from office.microsoft.com/powerpoint.
- . 2007. *Visio Home Page—Microsoft Office Online*, 2007. Available from office.microsoft.com/visio.
- Mitra, Nilo. *Simple Object Access Protocol (SOAP) 1.2: Primer*. W3C, June 24, 2003. Available from www.w3.org/TR/soap12-part0/.
- Modular Software-Programmable Radio Consortium 2001. *Software Communications Architecture Specification v2.2*, Specification, MSRC-5000SCA.
- Monroe, Robert T. 1998. Capturing Software Architecture Design Expertise With Armani, School of Computer Science, Carnegie Mellon University, technical report, CMU-CS-98-163.
- Moriconi, M., X. Qian, and R.A. Riemenschneider. 1995. Correct Architecture Refinement. *IEEE Transactions on Software Engineering* 21 (4): 356–372.
- Mos, A., and J. Murphy. 2004. COMPAS: Adaptive Performance Monitoring of Component-Based Systems. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, May (Edinburgh, Scotland).
- Mukerji, Jishnu, and Joaquin Miller, eds. 2003. *MDA Guide Version 1.0.1: Object Management Group*.
- NASA Jet Propulsion Laboratory. *Cassini-Huygens Mission to Saturn & Titan*. Available from saturn.jpl.nasa.gov/mission/index.cfm.
- Nelson, Harold G., and Erik Stolterman. 2003. *The Design Way: Intentional Change in an Unpredictable World—Foundations and Fundamentals of Design Competence*, Educational Technology Publications, Inc.
- Nemhauser, G. L., and L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. New York, NY, Wiley-Interscience.
- Nenas, Issa A.D., Reid Simmons, Daniel Gaines, Clayton Kunz, Antonio Diaz-Calderon, Tara Estlin, Richard Madison, John Guineau, Michael McHenry, I-Hsiang Shu, and David Apfelbaum. 2006. CLARAty: Challenges and Steps Toward Reusable Robotic Software. *International Journal of Advanced Robotic Systems* 3 (1): 23–30.
- Ng, Keng, Jeff Kramer, and Jeff Magee. 1996. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering* 3: 261–284.
- Nilsson, Nils J. 1980. *Principles of Artificial Intelligence*, Tioga Publishing Company.
- Norman, Donald A. 2002. *The Design of Everyday Things*. 1st Basic paperback ed. New York, Basic Books. Original edition, *The Psychology of Everyday Things*. New York: Basic Books, 1988.
- Nuseibeh, Bashar. 2001. Weaving Together Requirements and Architectures. *IEEE Computer* 34 (2): 115–117.
- Nuseibeh, Bashar, and Steve Easterbrook. 2000. Requirements Engineering: A Roadmap. *The Future of Software Engineering* (Limerick, Ireland), ACM Press, pp. 35–46.

- Nutaro, Jim. *Adevs: A Discrete Event System simulator*. Available from www.ornl.gov/~1qn/adevs/index.html.
- OASIS. *eXtensible Access Control Markup Language (XACML)*. OASIS, 2005. Available from docs.oasis-open.org/xacml/2.0/access-control-xacml-2.0-core-spec-os.pdf.
- . 2007. *Web Services Business Process Execution Language Version 2.0*, OASIS Standard.
- Object Management Group. 2003. *Meta Object Facility (MOF) Specification (1.4)* (PDF), April 2002. Available from www.omg.org/docs/formal/02-04-03.pdf.
- . 2003. *OMG XML Metadata Interchange (XMI) Specification (1.2)* (PDF), January 2002. Available from www.omg.org/docs/formal/02-01-01.pdf.
- . 2005. *UML Profile for DODAF/MODAF (UPDM) Request for Proposal*, OMG, RFP, OMG document syseng/05-08-01.
- , ed. 2001. *The Common Object Request Broker: Architecture and Specification*: Object Management Group.
- Oreizy, Peyman. 1998. Issues in Modeling and Analyzing Dynamic Software Architectures. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, June 30–July 3, (Marsala, Sicily, Italy).
- Oreizy, Peyman, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Greg Johnson, Nenad Medvidović, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14 (3): 54–62.
- Oreizy, Peyman, Nenad Medvidović, and Richard N. Taylor. 1998. Architecture-Based Runtime Software Evolution. In *Twentieth International Conference on Software Engineering (ICSE '98)*, April, (Kyoto, Japan), IEEE Computer Society, pp. 177–186.
- Oreizy, Peyman, Nenad Medvidovic, and Richard N. Taylor. 2008. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the Thirtieth International Conference on Software Engineering (Leipzig, Germany)*, ACM, pp. 899–910.
- Orfali, Robert, Dan Harkey, and Jeri Edwards. 1996. *The Essential Distributed Objects Survival Guide*. New York, NY, Wiley.
- Orso, A., D. Liang, M. J. Harrold, and R. Lipton. 2002. Gamma System: Continuous Evolution of Software after Deployment. In *International Symposium on Software Testing and Analysis (ISSTA 2002)*, July (Rome, Italy).
- Paczkowski, B. G., and T. L. Ray. 2004. Cassini Science Planning Process. In *SpaceOps 2004*, May 17–21, (Montreal, Canada).
- Parnas, David L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (12): 1053–1058.
- Perry, Dewayne E. 1987. Software Interconnection Models. In *Ninth International Conference on Software Engineering (ICSE)*, March, IEEE Computer Society, pp. 61–69.
- Perry, Dewayne E., and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17 (4): 40–52.
- Peter, Laurence J., and Raymond Hull. 1969. *The Peter Principle: Why Things Always Go Wrong*. New York, William Morrow & Company, Inc.
- Petroski, Henry 1985. *To Engineer Is Human*, St. Martin's Press.
- . 1992. *The Evolution of Useful Things*, Alfred A. Knopf, Inc.
- . 1994. *Design Paradigms: Case Histories of Error and Judgement in Engineering*, Cambridge University Press.
- . 1996. *Invention by Design: How Engineers Get From Thought To Thing*, Harvard University Press.

- Pezzè, Mauro, and Michal Young. 2007. *Software Testing and Analysis: Process, Principles, and Techniques*, John Wiley & Sons.
- Pfleeger, Charles P., and Shari Lawrence Pfleeger. 2003. *Security in Computing*, 3rd ed., Prentice Hall PTR.
- Pohl, Klaus, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed., New York, NY, Springer.
- Pollio, Marcus Vitruvius. 2007. *On Architecture* [Website (based on 1826 translation by Joseph Gwilt)], Jan 24, 2006. Available from penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/home.html.
- Polya, G. 1957. *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed., Princeton University Press. Original edition, 1945.
- Pooley, Rob. 2000. Software Engineering and Performance: A Roadmap. In *The Future of Software Engineering 2000* (Limerick, Ireland), ACM Press, pp. 189–200.
- Rabiner, Lawrence R. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE* 77 (2): 257–286.
- Raggett, Dave, Arnaud Le Hors, and Ian Jacobs. 1999. HTML 4.01 Specification, W3C. Available from www.w3.org/TR/REC-html40/.
- Rapanotti, Lucia, Jon Hall, Michael Jackson, and Bashar Nuseibeh. 2004. Architecture Driven Problem Decomposition. In *Twelfth IEEE International Requirements Engineering Conference (RE'04)*, September, 2004, (Kyoto, Japan), IEEE, pp. 80–89.
- Rational Software Corporation 2003. Rational Rose: Using Rose, IBM Corporation, Report number 800-024462-000.
- Reenskaug, Trygve. 1979. Thing-Model-View-Editor—An example from a Planning System, Xerox PARC, Technical note.
- . 2003. The Model-View-Controller (MVC): Its Past and Present. In *Java and Object-Oriented Software Engineering (JAOO)*, September 22–25, (Aarhus, Denmark).
- Reiss, Steven P. 1990. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software* 7 (4): 57–66.
- Ren, Jie, Richard Taylor, Paul Dourish, and David Redmiles. 2005. Towards An Architectural Treatment of Software Security: A Connector-Centric Approach. In *Workshop on Software Engineering for Secure Systems (SESS 05)*, in conjunction with ICSE 2005, May (St. Louis, MO).
- Ren, Jie, and Richard N. Taylor. 2003. Visualizing Software Architecture with Off-The-Shelf Components. In *Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, July 1–3, (San Francisco, CA), pp. 132–141.
- . 2005. A Secure Software Architecture Description Language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, Co-located with the twentieth IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), (Long Beach, CA).
- Robbins, Jason, N. Medvidović, David Redmiles, and David Rosenblum. 1998. Integrating Architecture Description Languages with a Standard Design Method. In *Twentieth International Conference on Software Engineering (ICSE)*, April, (Kyoto, Japan), ACM Press, pp. 209–218.
- Rosenblum, David S., and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the Sixth European Software Engineering Conference held jointly with the Fifth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 344–360, Zurich, Switzerland, September 1997.

- Roshandel, Roshanak, André van der Hoek, Marija Mikic-Rakic, and Nenad Medvidović. 2004. Mae: A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 13 (2): 240–276.
- Rozanski, Nick, and Eoin Woods. 2005. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional.
- Rumbaugh, James, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. 1990. *Object-Oriented Modeling and Design*. 1st ed., Prentice Hall.
- Sabbagh, Karl. 1989. *Skyscraper*, Viking.
- Saltzer, J. H., and M. D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proceedings of the IEEE* 63 (9): 1278–308.
- Schach, Stephen R. 2007. *Object-Oriented & Classical Software Engineering*. 7th ed., McGraw Hill Higher Education.
- Schneier, Bruce. 1995. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd ed., John Wiley & Sons.
- . 2000. *Secrets and Lies: Digital Security in a Networked World*, John Wiley & Sons, Inc.
- Schön, Donald. 1983. *The Reflective Practitioner: How Professionals Think in Action*. New York, Basic Books, Inc. Publishers.
- Seo, Chiyoung, Sam Malek, and Nenad Medvidović. 2007. An Energy Consumption Framework for Distributed Java-Based Systems. In Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE), November (Atlanta, GA), pp. 421–424.
- Sessions, R. 1997. *COM and DCOM: Microsoft's Vision for Distributed Objects*. New York, NY, John Wiley & Sons Inc.
- Shaw, M., R. DeLine, and G. Zelesnik. 1996. Abstractions and Implementations for Architectural Connections. In Third International Conference on Configurable Distributed Systems, May (Annapolis, MD), pp. 2–10.
- Shaw, Mary. 1994. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Studies of Software Design, Proceedings of a 1993 Workshop*, edited by D. A. Lamb: Springer-Verlag, pp. 17–32.
- Shaw, Mary, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. 1995. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* 21 (4): 314–335.
- Shaw, Mary, and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.
- Smolander, K. 2002. What is Included in Software Architecture? A Case Study in Three Software Organizations. In Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, pp. 131–138.
- Smolander, K., and T. Päiväranta. 2002. Describing and Communicating Software Architecture in Practice: Observations on Stakeholders and Rationale. In *Proceedings of the Fourteenth International Conference on Advanced Information Systems Engineering (CAISE'02)*, May 27–31, (Toronto, Ontario, Canada), Springer Berlin, pp. 117–133.
- Software Engineering Institute. *The Architecture Tradeoff Analysis Method (ATAM)*. Available from www.sei.cmu.edu/architecture/ata_method.html.
- Spencer, John. 2000. *Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools*, The Open Group, White Paper.
- Spitznagel, B., and D. Garlan. 2003. A Compositional Formalization of Connector Wrappers. In Twenty-Fifth International Conference on Software Engineering, pp. 374–384.
- Spivey, J. M. 1988. *Understanding Z*, Cambridge University Press.

- Stewart, W. J. 1994. *Introduction to Numerical Solution of Markov Chains*, Princeton University Press.
- Suryanarayana, Girish, Mamadou H. Diallo, Justin R. Erenkrantz, and Richard N. Taylor. 2006. Architecting Trust-Enabled Peer-to-Peer File-Sharing Applications. *ACM Crossroads*, 12 (4): 11–19.
- . 2006. Architectural Support for Trust Models in Decentralized Applications. In *Proceedings of the Twenty-eighth International Conference on Software Engineering*, May, (Shanghai, China), ACM Press, pp. 52–61.
- Suryanarayana, Girish, Justin R. Erenkrantz, and Richard N. Taylor. 2005. An Architectural Approach for Decentralized Trust Management. *IEEE Internet Computing*, November/December, pp. 16–23.
- SysML Partners. 2005. Systems Modeling Language (SysML) Specification version 0.9.
- SysML.org. 2007. SysML Partners (Webpage), 2007. Available from www.sysml.org/partners.htm.
- Szyperki, Clemens. 1997. *Component Software: Beyond Object-Oriented Programming*. New York, ACM Press.
- . 2002. *Component Software - Beyond Object-Oriented Programming*. 2nd ed., Addison-Wesley.
- Tanenbaum, Andrew S., and Maarten van Steen. 2002. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Prentice-Hall.
- Taylor, R., W. Tracz, and L. Coglianese. 1995. Software Development Using Domain-Specific Software Architectures. *Software Engineering Notes* 20 (5).
- Taylor, Richard N., Nenad Medvidović, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. 1996. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22 (6): 390–406.
- Taylor, Richard N., and Andre van der Hoek. 2007. Software Design and Architecture. The Once and Future Focus of Software Engineering, In *Future of Software Engineering (FOSE '07)*, edited by A. Wolf and L. Briand: IEEE Computer Society, pp. 226–243.
- Telecommunication Standardization Sector of ITU 2002. Specification and Description Language (SDL), ITU Standard Z.100.
- Telelogic AB. 2007. *Telelogic System Architect* (Website). 2007. Available from www.telelogic.com/Products/Systemarchitect/.
- Telelogic DOORS. Available from www.telelogic.com/products/doors/doors/index.cfm.
- The Globus Alliance. *The Globus Alliance* (Website). Available from www.globus.org/.
- The Omni Group. 2007. *The Omni Group—Applications—OmniGraffle*, 2007. Available from www.omnigroup.com/applications/omnigraffle/.
- The Open Group 2003. TOGAF (The Open Group Architecture Framework) Version 8.1 “Enterprise Edition,” The Open Group.
- . 2007. *Distributed Computing Environment Portal*, January 17 2005, Available from www.opengroup.org/dce/.
- Tichy, Walter F. 1985. RCS - A System for Version Control. *Software—Practice and Experience* 15 (7): 637–654.
- Tierney, B. 2000. A Monitoring Sensor Management System for Grid Environments. In *IEEE International Symposium on High Performance Distributed Computing*, August (Pittsburgh, PA).
- Tracz, W. 1993. LILEANNA: A Parameterized Programming Language. In *Proceedings of the Second International Workshop on Software Reuse*, pp. 66–78.
- Tracz, Will. 1994. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). *ACM SIGSOFT Software Engineering Notes* 19 (2): 52–56.

- . 1995. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. 1st ed, Addison-Wesley.
- . 1995. DSSA (Domain-Specific Software Architecture): Pedagogical Example. *ACM SIGSOFT Software Engineering Notes* 20 (3).
- Tracz, Will, Lou Coglianese, and Patrick Young. 1993. A Domain-Specific Software Architecture Engineering Process Outline. *ACM SIGSOFT Software Engineering Notes* 18 (2): 40–49.
- Tufte, Edward R. 1990. *Envisioning Information*, Graphics Press.
- . 2001. *The Visual Display of Quantitative Information*, Graphics Press.
- University of California, Berkeley. 1986. *UNIX Programmer's Manual*, 4.3 Berkeley Software Distribution, Computer Science Division, University of California–Berkeley.
- van der Hoek, André. 2000. Capturing Product Line Architectures. In *Fourth International Software Architecture Workshop (ISAW-4)*, June 4–5, (Limerick, Ireland), pp. 95–99.
- . 2004. Design-Time Product Line Architectures for Any-Time Variability. *Science of Computer Programming, Special Issue on Software Variability Management* 53 (30): 285–304.
- van Lamsweerde, Axel. 2000. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the Twenty-Second International Conference on Software Engineering*, (Limerick, Ireland), ACM Press, pp. 5–19.
- van Ommering, Rob. 2002. Building Product Populations with Software Components. In *Twenty-Fourth International Conference on Software Engineering (ICSE 2002)*, May 19–25, (Orlando, FL), pp. 255–265.
- van Ommering, Rob, Frank van der Linden, Jeff Kramer, and Jeff Magee. 2000. The Koala Component Model for Consumer Electronics Software. *IEEE Computer* 33 (3): 78–85.
- van Vliet, Hans. 2000. *Software Engineering: Principles and Practice*. 2nd ed., John Wiley & Sons, LTD.
- Vickers, Brian. 2004. Architecting a Software Architect. In *IEEE Aerospace Conference*, March 6–13, IEEE, pp. 4155–4161.
- Vogel, Ed. *Cigar Box Guitar. Make: Technology On Your Time*, 4, pp. 76–82.
- Vogels, W. 2003. Web Services Are Not Distributed Objects. *IEEE Internet Computing* 7 (6): 59–66.
- Warmer, Jos B., and Anneke G. Kleppe. 1998. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Object Technology Series, Addison-Wesley Professional.
- Wing, J. M. 2003. A Call to Action: Look Beyond the Horizon. *IEEE Security & Privacy*. 1 (6): 62–67.
- Woodside, Murray, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering In *Future of Software Engineering 2007*, edited by L. Briand and A. Wolf. IEEE-CS Press, pp. 171–187.
- Yellin, D. M., and R. E. Strom. 1994. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *OOPSLA '94*, October 1994, (Portland, OR).
- Zetter, Kim. *Kazaa Delivers More than Tunes*, Jan. 9, 2004. Available from www.wired.com/news/business/0,1367,61852,00.html.

Index

- AADL. *See* Architecture Analysis and Design Language
- Abbott, Russell J., 151
- abstraction, 87–90
 - components and, 69
- access control (architectural), 497–522
 - algorithm for checking, 502–505
 - connector-centric, 499–508
 - connectors' role in, 500–501
 - examples, 505–508
 - secure cooperation in, 505
 - Secure xADL and, 501–503
- access control lists (ACLs), 498
- access control models, 498–499
- Access Matrix Model, 498
- accuracy
 - in evaluation rubric, 198
 - precision v., 192–193
- ACLs (access control lists), 498
- Acme, 229–232
 - as architecture interchange language, 230, 232
 - evaluation rubric for, 230
 - Lunar Lander in, 230–232
- adaptation (architecture-centric), 525–564
 - APIs and, 552–553
 - architectural model and, 539
 - architecture-based
 - approaches, 555–557
 - architectures/styles and, 551–557
 - ArchStudio and, 550–551
 - autonomous, 557, 560–561
 - business case for, 562
 - change agents, 537–539
 - component/object
 - architecture approach, 553–554
 - components and, 475–476, 532–534
 - concepts of, 526–539
 - conceptual framework for, 539–542
 - configurations and, 477–478
 - connectors and, 476–477, 534
 - definition, 475, 525
 - erosion and, 532
 - event interfaces approach, 555, 556
 - on-the-fly, 525, 528–529, 557–560
 - patches/service
 - packs/upgrades/releases, 547
 - in physical building
 - architecture, 529–532
 - plug-ins approach, 553, 554
 - product-line motivations for, 527–528
 - scripting languages
 - approach, 555
 - shearing layers and, 529–532
 - sources/motivations for, 526–529, 536
 - structural elements subject to, 532–535
 - techniques, 542–561
 - timing and, 536–537
- adaptation management, 540, 541
- adaptor connectors, 71, 170–171, 172
- ADLs. *See* architecture description languages
- ADML. *See* Architecture Description Markup Language
- AEM (Architecture Evolution Manager), 550
- Aesop system, 177, 183, 325
- aesthetics
 - in visualizations, 261
 - visualizations rubric and, 272
- agency, 432–433, 444
- agency boundary, 421, 421n
- agile development process, turbine visualization of, 49–51
- Akamai, 422
- Albin, Stephen, 149
- Alexander, Christopher, 20, 22, 154, 155
- algebraic notations, 313
- Amazon.com, 511
- ambiguity, 192
 - in evaluation rubric, 198
- ambiguous models, 192, 193, 194
- American National Standards Institute (ANSI), 622
- analogies, 146–147
- analysis (architectural), 291–336
 - in architecture-driven deployment, 396–399
 - of architectures, at different abstraction levels, 308–310

- analysis (*Continued*)
 ATAM and, 314, 320–322
 automation levels, 314–315
 business case for, 334
 comparison of architectures
 in, 310
 compatibility in, 302–303
 completeness in, 295–296
 component-level, 304–305
 concerns, 310–312
 connector-level, 39–40,
 304–305
 consistency in, 296–302
 correctness in, 303
 definition, 291
 dimensions, 318
 dynamic, 313–314
 of exchanged data, 307–308
 goals of, 295–303
 inspections and reviews,
 317–322
 manual, 314–315
 model-based, 38–40, 291,
 322–327
 partially automated, 315
 scenario-based, 314
 scope of, 303–310
 simulation-based, 328–333
 stakeholders and, 315–317
 static, 313
 subsystem-level, 305–307
 system-level, 305–307
 techniques, 317–333
 testing and, 38–40
 types of, 313–314
 XTEAM and, 330–333
 Anna, 313
 ANSI (American National
 Standards Institute),
 622
 ANSI/IEEE *Standard*
 1471–2000, 68
 Apache Architecture
 Recovery Project, 143
 Apache Derby, 174
 Apigen (xADL data binding
 library generator), 235
 APIs. *See* application
 programming
 interfaces
 Apollo missions to moon, 99,
 155
 application domains. *See*
 domains
 Application layer, 516, 517
 application programming
 interfaces (APIs),
 552–553
 application-independent
 property, of connectors,
 71, 159–160, 182
 applied architectures/styles,
 413–446. *See also*
 architecture; styles
Applied Cryptography
 (Schneier), 496
 arbitrator connectors, 170,
 171
De Architectura (Vitruvius), 3
 ArchDiff, 550
 ArchEdit, 285
 Archipelago, 237, 240,
 285–286
 Lunar Lander in, 240,
 258–259
 architects (software), 655–674
 bad habits 661–663
 challenges for, 673
 connectors and, 158
 as domain experts, 658–659
 qualifications, Vitruvius
 and, 3–4
 relations with other
 stakeholders, 664,
 669–672
 skills of, 657–658, 673, 674
 as software designers, 658
 as software engineering
 economists, 661
 as software technologists,
 659–660
 as stakeholders, 79,
 316
 as standards compliance
 experts, 660–661
 tasks of, 663–665
 teams, 665–669
architectural
 access control. *See* access
 control
 analysis. *See* analysis
 basic concepts. *See*
 terminology
 conception. *See* conception
 configurations. *See*
 configurations
 degradation. *See*
 degradation
 design. *See* design
 drift. *See* drift
 erosion. *See* erosion
 mismatches. *See* mismatches
 modeling. *See* modeling
 models. *See* models
 patterns. *See* patterns
 perspectives. *See*
 perspectives
 recovery. *See* recovery
 stakeholders. *See*
 stakeholders
 styles. *See* styles
 Architectural Trade-Off
 Analysis Method
 (ATAM), 314,
 320–322
 architecture(s) (software)
 abstraction levels, analysis
 of, 308–310
 as abstractions of systems,
 191–192
 adaptation and. *See*
 adaptation
 applied. *See* applied
 architectures/styles
 Brooks' Law and, 51–52
 building architecture *v.* *See*
 architecture (of
 buildings)
 characteristics, 310–312
 configuration management
 of, 601–602
 decentralized, 424–433
 definitions, 58, 67–68, 81,
 185
 deployment and,
 387–405
 descriptive, 59–63
 designing of. *See* design
 on the desktop, 12–15
 as discipline, 4

- domain-specific. *See*
 - domain-specific software architectures
- as [elements, form, rationale], 67–68
- fundamental understandings of, 24–25
- implementation and, 33–38
- mobility and, 405–408
- prescriptive, 59–61
- principal design decisions and, 1, 20, 58, 185, 189, 191, 192, 209
- processes v., 4, 42, 76–77, 79
- product lines and, 15–19, 591–592
- reference, 58, 93, 572
- requirements engineering v., 150
- robotic, 438–443
- service-oriented, 433–436
- software engineering and, 23–55
- from specific domains, 437–443
- styles. *See* styles
- teams, 665–669
- terminology, 58–76, 80
- architecture (of buildings)
 - analogy, 2–6
 - designs 3, 83
 - as discipline, 485
 - How Buildings Learn* (Brand), 22
 - A Pattern Language: Towns, Buildings, Construction*, 155
 - security and, 487
 - shearing layers, 529–532
 - skyscraper construction, 22
 - The Timeless Way of Building* (Alexander), 20, 22, 154–155
 - tract houses, DSSAs as, 93–94
 - Vitruvius on, 3–4, 5, 20, 91
- Architecture Analysis and Design Language (AADL), 221, 225–228
- ADML v., 644–645
- components, 225–226
- evaluation rubric for, 226
- Lunar Lander in, 226–228
- support for, 226
- architecture description languages (ADLs), 76, 209–242
- AADL, 221, 225–228
- Acme, 229–232
- ADML, 232–234
- Darwin, 210–213, 221, 244
- domain-specific/style-specific, 221–228
- early, 209–221
- extensible, 229–242, 244
- Koala, 16–19, 22, 221–222
- Rapide, 185, 213–219, 268
- requirements for, 209
- UML as, 209
- Weaves, 221, 222–225
- Wright, 219–221
- xADL, 234–242
- Architecture Description Markup Language (ADML), 232–234
- AADL v., 644–645
- Lunar Lander in, 233–234
- Architecture Evolution Manager (AEM), 550
- architecture implementation
 - frameworks, 338, 340–341, 350–359
 - building, 346–347
 - C2 frameworks 353–354
 - defined, 340
 - efficiency and, 343
 - evaluating, 342–343
 - fidelity in, 342
 - java.io framework, 351–352
 - matching assumptions in, 342–343
 - middleware and, 343–346. *See also* middleware
 - for pipe-and-filter style, 350–352
 - platform support, 342
 - selecting, 343
 - standard I/O framework, 341, 350–351
- architecture interchange language, Acme as, 230, 232
- architecture-centric adaptation. *See* adaptation
- architecture-driven
 - deployment, 387–405
 - implementation in, 399–402
 - modeling in, 394–396
 - planning in, 388–394
 - tool-support, 402–405
- architecture-to-implementation
 - consistency, 349–350, 377, 539
- ArchStudio environment, 242, 522, 550–551
- adaptation and, 550–551
- ArgoUML, 276, 277
- artificial intelligence
 - applications, 112
- The Art of Software Architecture* (Albin), 149
- The Art of Systems Architecting* (Maier and Reichtin), 187, 246
- “As We May Think” (Bush), 29
- ATAM. *See* Architectural Trade-Off Analysis Method
- ATM (automated teller machine) system, 108
- attacks. *See* threats
- audit trail, 492
- authentication, 491, 492, 515
 - signature-based, 515
- automation levels, analysis, 314–315
- autonomic computing, 561
- autonomous adaptation, 557, 560–561
- availability
 - definition, 478
 - security and, 492

- Avionics Architecture
 - Description Language, 225. *See also*
 - Architecture Analysis and Design Language
- axiomatic notations, 313
- bad habits (architects)
 - delusions of grandeur, 663
 - inflexibility, 662
 - isolationism, 662–663
 - micromanagement, 662
 - perfectionism, 661–662
 - ultimate bad habit, 663
- batch-sequential style, 109–110, 140
 - Lunar Lander in, 110
 - pipe-and-filter v., 607
- bbFTP, 175
- BBoM (Big Ball of Mud) style, 137
- behavioral characteristics, of software architecture, 311
- behavioral constraints, 189
- behavioral inconsistencies, in models, 298–299
- behavioral modeling, 189
- behavioral viewpoints, 196
- behaviors, implementation mapping for, 339–340
- Berners-Lee, Tim, 22
- better-cheaper-faster—pick any two!, 398–399, 483
- Big Ball of Mud (BBoM) style, 137
- binary integer programming (BIP), 410
- bindings, 556
- BIP (binary integer programming), 410
- BitTorrent, 176, 432
- blackboard style, 112, 141
 - Lunar Lander in, 112, 113
- blamestorming, 147
- Boehm, Barry, 648
- Booch diagrams, 203
- Booch, Grady, 151
- bottom-up design, 154, 367n
- Boyd, John, 55, 145–146
 - Boyd: the Fighter Pilot Who Changed the Art of War* (Coram), 55
- brainstorming, 147
- Brand, Stewart, 22, 529
 - How Buildings Learn*, 22, 529, 530
 - shearing layers and, 529–532
- Brooks, Fred, 51, 52, 669, 674
- Brooks' Law, 51–52
- browsers. *See also* Firefox
 - extension of functionality in, 528–529
- brute force attack, 494
- buildings, architecture of. *See* architecture
- Bush, Vannevar, 29n
- business case
 - for adaptation, 562
 - for analysis, 334
 - for architectural terminology, 80
 - for architecture-centric development, 53
 - for connectors, 182
 - for deployment and mobility, 408–409
 - for design, 153
 - for DSSE, 616–618
 - for implementation, 377–378
 - for modeling, 245
 - for security, 522–523
 - for standards, 652–653
 - for trust management, 523
 - for visualizations, 289
- business product lines, 592–593
- C++, 102
- C programming language, 102
 - stdio and, 350–351
- C2 frameworks
 - Flexible, 355–359
 - Lightweight, 353–354, 358–359, 366–376
- C2 (components and connectors) style, 124, 126–131, 142
 - architecture implementation frameworks for, 352–359
 - benefits, 126
 - as combination of styles, 138
 - compositionality and, 127
 - constraints, 126–127, 128
 - distributed objects style v., 126
 - KLAX-like game in, 129–131
 - Lunar Lander in, 127–128, 366–367
 - PACE and, 517. *See also* PACE style
- Cameron, John, 151
- canonical requirements, DSSA, 585–588
- canonical visualizations, 250–251
- cargo routing application, 60–61, 62, 63
 - degradation in, 64
 - deployment perspective, 65, 66
 - drift in, 64
 - erosion in, 65
 - implementation-level view, 77, 78
 - prescriptive/descriptive architectures, 60–61, 62, 63
- CBSE (Component-Based Software Engineering), 81
- centralized reputation-based systems, 511
- change, 525. *See also* adaptation
 - corrective, 526
 - 'designing for change,' 561
- change agents, 537–539
- channels. *See* ducts
- Chapman, Merrill R., 663

- characteristics, 310–312
- chartjunk, 263
- cipher, 490
- classification (connectors), 184
 - framework, 162, 163
 - types, 164–165
- ClearCase, 601
- Clements, Paul, 606
- client-server style, 108–109, 140
 - ATM system and, 108
 - Lunar Lander in, 108–109
- client-server-based data
 - distribution
 - connectors, 175
- clustering, 142–143
- coblalberation, 147
- Code Complete (Microsoft), 85
- code folding, 273
- code mobility, 405–408
- code-on-demand, 406
- collusion, 514, 519
- COM (Component Object Model), 554
- combination of styles/patterns, 138–139
- Communicating Sequential Processes (CSP)
 - formalism, 219, 220, 313
- communication services, 163
 - connectors and, 163
- comparable trust, 515
- compatibility
 - in analysis, 302–303
 - of connectors, 176, 177, 181
- compatibility matrix, 180
 - connector, 178–179, 180–181
- complete mediation, principle of, 494
- completeness, in analysis, 295–296
- complex modeling, 188–189, 194–198, 242–243
- complex styles, 124, 126, 142
- complexity, 459–467
 - components and, 460–463
 - configurations and, 464–467
 - connectors and, 463–464
 - definitions, 459
- Component Object Model (COM), 554
- component security
 - architecture, Firefox, 507–508
- Component-Based Software Engineering (CBSE), 481
- component-level analysis, 304–305
- components, adaptation and, 475–476, 532–534
- components (software), 68–70
 - abstraction and, 69
 - adaptation and, 475–476, 532–534
 - application-specific, 69
 - complexity and, 460–463
 - definitions, 69, 70
 - dependability and, 480
 - efficiency and, 451–452, 453
 - encapsulation and, 69
 - implementation mapping
 - for, 339
 - modeling and, 188
 - modularity and, 69
 - PACE, 518
 - reuse of, 69
 - scalability and, 468–471
- components and connectors
 - style. *See* C2 style
- composite connectors, 162–163, 165, 177, 180
- composite distribution
 - connectors, 172–173
- composite structure diagram, UML, 255
- compositionality, 127
- comprehensibility
 - in visualizations, 260
 - visualizations rubric and, 272
- conception (architectural), 87–91
 - in absence of experience, 144–149
 - tools for, 87–91. *See also* refined experience
- conceptual framework for
 - architectural adaptation, 539–542
- concurrency
 - constraints, 190
 - implementation and, 347–348
 - modeling, 190
 - viewpoints, 196
- Confessions of a Used Program Salesman* (Tracz), 619
- configuration management, of architectures, 601–602
- configurations (architectural)
 - adaptation and, 477–478
 - complexity and, 464–467
 - definition, 72
 - dependability and, 481–483
 - efficiency and, 455–459
 - implementation mapping
 - for, 339
 - modeling and, 188
 - scalability and, 473–475
- connector-level analysis, 304–305
- connectors (software), 70–71, 157–184
 - adaptation and, 476–477, 534
 - adaptor, 71, 170–171, 172
 - analysis of, 39–40
 - application-independent
 - property of, 71, 159–160, 182
 - arbitrator, 170, 171
 - architectural access control
 - and, 500–501
 - business case for, 182
 - classification, 162–165, 184
 - client-server-based data
 - distribution, 175

connectors (*Continued*)

communication services,
163
compatibility matrix,
178–179, 180–181
compatibility of, 176, 177,
181
complexity and, 463–464
composite, 162–163, 165,
177, 180
conversion services, 164
coordination services,
163–164
CORBA and, 157, 175
data access, 167–168
data-intensive. *See* distribution
connectors
DCOM and, 157
definition, 70, 157
dependability and, 480–481
distribution. *See* distribution
connectors
distributor, 171–172
ducts and, 160, 161, 168,
181, 182
efficiency and, 453–455
event, 166–167
event-based data
distribution, 161,
173–174
explicit focus on, 181, 182,
183–184
facilitation services,
164
as first-class entities, 181,
184
foundations, 161–163
grid-based data distribution,
174–175
implementation mapping
for, 339
linkage, 168–169
mismatches and, 158, 163,
164, 180–181
modeling and, 188
P2P-based data distribution,
176
procedure call, 71, 165–166
purpose of, 157
RMI and, 157

roles, 163–164
scalability and, 471–473
selecting, 177, 180
service classes, 163–164
simple, 164, 165
stream, 111, 169–170
transfer of control, 157, 163,
182
transfer of data, 163, 165,
185
types, 164–165
variation dimensions of,
164–165, 166, 167,
168, 169, 170, 171,
172, 181
consistency. *See also*
inconsistencies
in analysis, 296–302
architecture-to-
implementation,
349–350, 377, 539
modeling rubric and, 198
among views, 196–198
visualizations and, 259–260
visualizations rubric and,
272
consistency maintenance,
540
constraints
behavioral, 189
of C2 style, 126–127, 128
concurrency, 190
interaction, 189
of styles, 5, 138
for UML, 204
consumer electronics,
608–611. *See also* Koala
context information diagram,
579, 580
contract, security, 500, 501
control-flow diagrams, 585,
586
conversion services,
connectors and, 164
coordination services,
connectors and,
163–164
Coram, Robert, 55
CORBA, 37, 132–137. *See also*
distributed objects style

connectors and, 157, 175
designing and, 91
Lunar Lander training
version in, 135, 136
as middleware, 37
SCA and, 612, 613
SDRs and, 151
corrective change, 526
correctness, in analysis, 303
cryptography, 490–491
public key, 490
shared key, 490
CSP (Communicating
Sequential Processes)
formalism, 219, 220,
313
customer needs statement,
587
customers, as stakeholders,
79, 316–317
CVS, 601
cyclic approach (design
strategy), 86
DAC (discretionary access
control) models, 498
Darwin, 210–213, 244
evaluation rubric for,
210
Koala and, 221. *See also*
Koala
loops, 211
Lunar Lander in, 210–211,
212
ports, 210
data access
persistent, 167
transient, 167
data access connectors,
167–168
data binding library (xADL),
235
data distribution connectors.
See distribution
connectors
data-flow diagram, 584, 585
data-flow styles, 109–111,
140
batch-sequential, 109–110,
140, 607

pipe-and-filter, 14–15,
 110–111, 140, 160,
 222, 350–352,
 360–366, 607
 data-intensive connectors. *See*
 distribution connectors
 DCOM (Microsoft), 37, 133,
 157, 343, 491–492, 501
 authentication and, 491,
 492
 connectors and, 157
 de facto standards, 622,
 625
 de jure, 622, 625
 Death Star, Web services
 (WS-*), 436
 decentralized architectures,
 424–433
 grid computing, 425–426
 decentralized reputation-based
 systems, 511. *See also*
 XREP
 decentralized trust
 management
 architectural approach to,
 514–522
 PACE and, 517–522
 decisions. *See* design decisions
 defense in depth, principle of,
 496–497
 definitions. *See* terminology;
 specific definitions
 degradation (architectural),
 61–64, 77
 DeLine, Robert, 171
 delusions of grandeur, 663
 denial of service, 514,
 519
 dependability, 478–483
 deployment
 architecture-driven,
 387–405
 challenges, 385–387
 mobility and, 381–387,
 408–409
 perspective, 65, 66, 67
 process, 400–402
 viewpoints, 195
 Derby (Apache), 174
 descriptive architecture, 59–60
 cargo routing example,
 60–61, 62, 63
 DeSi, 404, 410
 design (architectural), 29–31,
 83–155
 books on, 154–155
 bottom-up, 154, 367n
 Boyd on, 145–146
 buildings and, 3, 83
 business case for, 153
 conception. *See* conception
 definition, 29
 design phase v., 29–30
 disciplined study of, 83–84
 first step. *See* conception
 implementation and,
 151–152
 OOD and, 31–32
 operating systems, layered
 style and, 106
 recovery. *See* recovery
The Reflective Practitioner
 (Schön), 55, 155
 requirements engineering
 and, 25–29, 150–151
 by simulation, 151
 software reuse and,
 152
 stepwise refinement, 89,
 154
 techniques, 30, 31–33. *See*
 also domain-specific
 software architectures
The Timeless Way of Building
 (Alexander), 20, 22,
 154–155
 top-down, 148, 154
 unprecedented, 144–149
 design decisions, 58
 functional behavior, 58
 implementation, 58
 interaction, 58
 in modeling of styles,
 189–190
 nonfunctional properties,
 58
 principal
 architecture and, 1, 20,
 58, 185, 189, 191, 192,
 200, 209

structure, 58
 styles and, 73, 99, 189
 design principles, for security,
 492–497
 design processes, 85–86,
 149–152
 JSD method, 85, 151
 management of, 86
 spiral model, 42, 53, 85,
 648
 stages in, 85
 strategies, 85–86
 ‘underline the nouns,
 underline the verbs’
 approach, 151
 waterfall model, 25, 30, 42,
 85
 design rationale,
 implementation
 mapping and, 339
 ‘designing for change,’ 561
The Design of Everyday Things
 (Norman), 55
 desktop, architecture on the,
 12–15
 “Destruction and Creation”
 (Boyd), 145, 146
 Deutsch, Peter, 415
 developers (software), as
 stakeholders, 79
 concerns of, 316
 development processes
 (software), 2, 42–52,
 76–77
 architecture v., 4, 25, 42,
 76–77, 79
 architecture-centric, 1, 20,
 23–53
 turbine visualization, 43–51
 Twin Peaks model, 51, 55
 waterfall model, 25, 30, 42,
 85
 diagrams. *See* Unified
 Modeling Language;
 specific diagrams
 digital signatures, 490, 519
 Dijkstra, Edsger, 6
 dining philosophers problem,
 282–283
 direct analogies, 146

- direct inconsistencies, 196
- disciplined study, of design, 83-84
- discourse levels/terms,
 - abstraction and, 89-90
- discrete-time Markov chains (DTMCs), 327
- discretionary access control (DAC) models, 498
- distributed applications, 414-416
 - limitations of, 415
- distributed computing
 - fallacies of, 415
 - grid computing, 425-426
- distributed hypermedia
 - application, 417, 424
- distributed objects style, 124, 126, 142
 - CORBA. *See* CORBA
 - Engineering Distributed Objects* (Emmerich), 155
 - RMI and, 133
- distribution connectors
 - (data-intensive connectors), 71, 172-173
 - client-server-based, 175
 - composite, 172-173
 - event-based, 161, 173-174
 - grid-based, 174-175
 - P2P-based, 176
- distributor connectors, 171-172
- divergence step, 144-145
- DNS. *See* domain name service
- document type definitions (DTDs), 233
- DoDAF, 194, 253, 628-634
 - Lunar Lander and, 630-633
- domain dictionary, 577, 578
 - elements of, 577
 - feature model, 577-587
- domain name service (DNS), 172
- domains
 - DSSE and, 568, 569, 573
 - subdomains and, 566, 567
- domain-specific ADLs, 221-228
- domain-specific software
 - architectures (DSSAs), 29, 32-33, 93-94, 574-608
 - patterns v., 94
 - product lines, styles, and, 606-608
 - reference architectures and, 588-591
 - reference requirements, 585-588
 - turbine visualization technique and, 49
- domain-specific software engineering (DSSE), 565-619
 - business case for, 616-618
 - business goals and, 568, 569, 573
 - defined, 568
 - domains and, 568, 569, 573
 - examples, 608-615
 - problem spaces, 567, 570, 571, 572
 - reuse and, 616-617
 - solution spaces, 570, 571, 572
 - technology and, 568, 569, 573
- drift (architectural), 64
- DSSAs. *See* domain-specific software architectures
- DSSE. *See* domain-specific software engineering
- DTDs (document type definitions), 233
- DTMCs (discrete-time Markov chains), 327
- ducts, 160, 161, 168, 181, 182
- dynamic analysis, 313-314
- dynamic aspects of system models, 190-191
 - evaluation rubric and, 198
- dynamic models, 190, 260
 - evaluation rubric and, 198
- dynamic properties,
 - implementation mapping for, 339-340
- dynamism
 - in visualizations, 260
 - visualizations rubric and, 272
- EASEL project, 263, 264
- eBay, 511
- Eclipse software development environment, 226
- ECMAScript, 622
- economy of mechanism,
 - principle of, 494
- effect visualizations, 266, 268
- efficiency
 - architecture implementation frameworks and, 343
 - components and, 451-452, 453
 - configurations and, 455-459
 - connectors and, 453-455
 - definition, 450
 - [elements, form, rationale], architecture as, 67-68
- emergency response systems (ERS), 384, 390
 - candidate deployment views, 397, 398
 - deployments, 391
 - hardware configuration of, 395
 - latency, durability, interaction volume, 393
 - software architectural configuration of, 395
- Emmerich, Wolfgang, 155
- encapsulation, modularity and, 69

- encryption, 488, 490
- Engineering Distributed Objects* (Emmerich), 155
- engineering product lines, 593–594
- entity-relationship (ER) diagram, 270, 579
- Envisioning Information* (Tufte), 263
- ER (entity-relationship) diagram, 270, 579
- erosion (architectural), 64–65
 - adaptation and, 532
- evaluation rubrics. *See* rubrics
- event connectors, 166–167
- event interfaces, 555, 556
- event-based data distribution connectors, 161, 173–174
- event-based styles, 120–123, 141
 - Lunar Lander in, 122, 303
- event-buses, 161
- events (defined), 166
- evolution management, 541
- evolution/maintenance (software), 40–42
- The Evolution of Useful Things* (Petroski), 27
- exclusive resources, 388
- experience. *See* patterns; refined experience; styles
- expert systems. *See* rule-based/expert systems
- extensibility
 - of visualizations, 261
 - visualizations rubric and, 272
- eXtensible Access Control Markup Language (XACML), 501, 502
- extensible ADLs, 229–242
 - Acme, 229–232
 - ADML, 232–234
 - xADL, 234–242
- Extensible Markup Language. *See* XML
- eXtensible Tool-chain for Evaluation of Architectural Models (XTEAM), 330–333
- facilitation services, connectors and, 164
- fail-safe defaults, principle of, 493–494
- “Fallacies of Distributed Computing” (Deutsch), 415
- families. *See* product lines
- fault-tolerance, 478–479
- FDR (Formal Systems Ltd.), 219
- feature model, 580–582
 - feature relationship diagram, 582
 - representation model, 583, 584
 - use-case diagram, 583
- feature relationship diagram, 582
- fidelity
 - in architecture implementation frameworks, 342
 - in visualization techniques rubric, 272
 - visualizations and, 259
- Field Programmable Gate Arrays (FPGAs), 611, 611n, 612
- Fielding, Roy, 22, 416
- filter programs, 13, 14
- financial markets, event-based styles and, 121, 122
- finite state processes (FSP), 282
- Firefox, 505–508
- first-class entities, connectors as, 181, 184
- flat model (team structure), 669
- Flexible C2 Framework, 355–359
- FMC (Fundamental Modeling Concepts), 143
- formal modeling techniques, 244
- formal models, 313
- FPGAs (Field Programmable Gate Arrays), 611, 611n, 612
- frameworks, 35–37. *See also* architecture implementation frameworks
 - conceptual, for adaptation, 539–542
 - development, for robotic systems, 442–443
 - Java Swing, 36–37, 69, 96
 - MFC, 36–37
- fraudulent actions, 514, 519
 - on eBay, 511
- free-form/informal modeling techniques, 244
- FSP (finite state processes), 282
- functional aspects of systems
 - design decisions and, 58
 - models of, 191
 - non-functional v., 191
- functional v. non-functional aspects inconsistencies, 197–198
- functionality-scalability-performance—pick any two!, 399
- Fundamental Modeling Concepts (FMC), 143
- Gambetta, Diego, 509, 510
- GAMMA, 411
- generative technologies, 35, 348–349
 - generation, 348–349
 - MDA and, 348, 379, 650–651
- GIL, 313
- GLIDE, 174

- Globus Grid
 - recovered architecture of, 426
 - Toolkit, 175
- Gnutella
 - modeling and, 243
 - peer-to-peer
 - communication and, 123, 427–429, 512
- Google, 147, 422–424
- goto* statement considered harmful, 6
- Granatelli, Andy, 522
- graphical modeling
 - (PowerPoint-style modeling), 186, 201–203
 - evaluation rubric for, 201–202
 - Lunar Lander in, 202–203, 292
 - OmniGraffle and, 201, 251, 255
- graphical visualizations, 253–255, 274–276
 - evaluation rubric for, 275–276
- greenfield development, 29
- grid computing, 425–426
- Grid Metadata Catalog Service (MCS), 175
- Grid Security Infrastructure (GSI), 174
- grid-based data distribution connectors, 174–175
- GridFTP software, 175
- GSI. *See* Grid Security Infrastructure
- Hasso-Plattner-Institute for Software Systems Engineering, 143
- heap/stack memory access, 167
- heterogeneity, definitions, 468
- Hidden Markov Model (HMM), 327
- hierarchical composition, 137, 467
- hierarchical model (team structure), 669
- higher-order connectors. *See* composite connectors
- HMM (Hidden Markov Model), 327
- honey-baked ham syndrome, 306–307
- Honeywell, 225
- How Buildings Learn—What Happens After They're Built* (Brand), 22, 529, 530
- How to Solve It* (Polya), 26, 55
- HTTP protocol, Web and, 95, 243
- hybrid client-server/P2P, 426–427
- hybrid product lines, 606
- hybrid visualizations, 255, 256
- hypertext, invention of, 29n
- IDEO design firm, 154
- IDL (Interface Definition Language), 133, 134, 344
- IEEE
 - standard 1471, 68, 627–628
- IETF (Internet Engineering Task Force), 22
- IIS. *See* Microsoft Internet Information Service
- "I'll know it when I see it" (IKIWISI), 655
- impersonation, 514, 519
- implementation, 337–379
 - architecture and, 33–38
 - in architecture-driven deployment, 399–402
 - architecture-to-implementation consistency, 349–350, 377, 539
 - business case for, 377–378
 - concurrency and, 347–348
 - design and, 151–152
 - in design decisions and, 58
 - generative technologies and, 35, 348–349
 - Lunar Lander, 360–376
 - mapping problem, 338–339
 - mappings, 339–340
 - of product lines, 602–604
 - strategies, 35–38. *See also* middleware; remote procedure calls
- implicit invocation, 123
- implicit invocation styles, 118–123, 141
 - event-based, 120–123, 141
 - publish-subscribe, 118–120, 121, 141
- In Search of Stupidity* (Chapman), 663
- inconsistencies
 - in models, 296–302
 - among views, 196–198
- incremental approach (design strategy), 86
- inflexibility, 662
- informal graphical techniques. *See* graphical modeling; graphical visualizations
- informal models, 312
- informal terms, terminology v., 57
- informal/free-form modeling techniques, 244
- information caching, 167
- information model, 577, 579–580
 - context information diagram, 579, 580
 - ER diagram, 579
 - object diagram, 579–580, 581
- inspections, 317–322
- interprocess communication (IPA) mechanism, 134
- interaction characteristics, of software architecture, 311–312
- interaction constraints, 189
- interaction inconsistencies, in models, 299–300
- interconnection models, software, 168, 184
- Interface Definition Language (IDL), 133, 134, 344

- interfaces
 - definition, 188
 - implementation mapping for, 339
 - mismatches and, 72
 - modeling and, 188
- International Organization for Standardization (ISO), 622
- Internet Engineering Task Force (IETF), 22
- Internet-scale applications, 422–424. *See also* Web Akamai, 422
 - Google, 147, 422–424
 - modeling of, 243
- interpreter styles, 115–118, 141
 - Lunar Lander in, 116, 117
 - mobile code, 116–118, 119, 141, 155
- IPA (interprocess communication) mechanism, 134
- ISO (International Organization for Standardization), 622
- isolationism, 662–663
- Jackson System Design (JSD) method, 85, 151
- JAMM, 411
- Java, 102
 - designing and, 91
 - method calls, 157, 158, 165
 - RMI, 133, 157, 175, 344, 378, 379
 - Swing framework, 36–37, 69, 96
- Java Native Interface (JNI), 346
- java.io framework, 351–352
 - implementing Lunar Lander in pipe-and-filter style with, 360–366
- JavaScript, 505–506
 - browsers and, 529n
- JNI (Java Native Interface), 346
- Jones, J. Christopher, 144, 145, 146, 148
- JSD (Jackson System Design) method, 85, 151
- JXTA-based data distributors, 176
- Kazaa, 512
- KISS (Keep it Simple and Small), 494
- KLAX, 129n
- KLAX-like video game, 129–131. *See also* C2 style
- Koala, 221–222
 - consumer electronics and, 608–611
 - Darwin and, 221
 - evaluation rubric for, 222
 - product lines and, 16–19
 - reusable assets and, 17, 18, 19
- Kramer, Jeff, 559
- Labeled Transition State Analyzer (LTSA), 282–284
 - dining philosophers problem, 282–283
 - effect visualizations, 268
 - evaluation rubric for, 284
- language-influenced styles, 102–104, 140
 - main program and subroutines, 102–103, 140
 - object-oriented, 103–104, 140
- languages. *See* modeling techniques; natural language; programming languages
- large files transfer protocol, 175
- latency, 432–433, 444
- layered style, 104–109, 140
 - client-server, 108–109, 140
 - operating system design and, 106
 - REST and, 418–419
 - virtual machines, 105–108, 140
- layering, 263
- layers
 - shearing, 529–532
 - in software system, 530n. *See also* layered style
- leaders, architects as, 664–665
- least common mechanism, principle of, 495
- least privilege, principle of, 493
- least-common-denominator problem, 624
- Lego blocks, 14, 17
- Lego Mindstorms, 443
- life cycle. *See* development processes
- Lightweight C2 Framework, 353–354, 358–359
 - classes and relationships from, 354
 - Flexible C2 v., 358–359
 - Lunar Lander in C2 style implemented with, 366–376
- linkage connectors, 168–169
- Linux, 154
 - architecture of, 465, 466
 - high level model of, 300, 301, 302
 - Process Scheduler connector, 300, 301, 302
 - recovery project and, 143
- literature searching, 147–148
- load balancing, 473
- logical mobility, 405
- logical viewpoints, 195
- LTSA. *See* Labeled Transition State Analyzer
- Lunar Lander
 - in AADL, 226–228
 - in Acme, 230–232
 - in ADML, 233–234
 - in Archipelago, 240, 258–259
 - in batch-sequential style, 110

- Lunar Lander (*Continued*)
- in blackboard style, 112, 113
 - in C2 style, 127–128, 367
 - in client-server style, 108–109
 - context information
 - diagram and, 580
 - control-flow diagram and, 586
 - in CORBA, 135, 136
 - customer needs statement and, 587
 - in Darwin, 210–211, 212
 - data-flow diagram and, 585
 - DoDAF and, 630–633
 - domain dictionary for, 578
 - efficiency NFP and, 447
 - elements, hybrid
 - visualization of, 255, 256
 - ER diagram and, 581
 - in event-based style, 122, 303
 - feature relationship diagram and, 582
 - graphical visualizations of, 253, 254
 - implementation, 360–376
 - in interpreter style, 116, 117
 - in *main program and subroutine* style, 102–103
 - mandatory requirements, 588
 - in mobile code style, 118, 119
 - modeling techniques and, 199
 - in natural language, 200–201
 - object diagram and, 581
 - in object-oriented style, 104, 105
 - optional requirements, 588
 - in P2P style, 124, 125
 - in pipe-and-filter style, 111
 - in PowerPoint, 202–203, 292
 - product line of, 601
 - in publish-subscribe style, 120, 121
 - in Rapide, 215–218, 281, 293
 - reference architecture for, 591
 - reference requirements and, 588
 - reliability NFP, 447
 - representation model and, 584
 - in rule-based style, 114–115
 - SCC applied to, 98–99
 - state-transition diagram and, 587
 - styles and, 101–102
 - SysML parametric diagram for, 643
 - SysML requirements diagram for, 642
 - in UML, 206–209
 - use-case diagram and, 583
 - variable requirements, 588
 - views of, 195, 197
 - in virtual machines style, 107–108
 - in Weaves, 224–225
 - in Wright, 219–220
 - in xADL, 237–242
 - in xADLite, 258–259, 304
- MAC (mandatory access control) models, 499
- Magee, Jeff, 559
- Maier, Mark, 187, 246
- main program and subroutines (style), 102–103, 140
- Lunar Lander and, 102–103
- maintenance. *See* evolution/maintenance
- managers (software), as stakeholders, 79
- concerns of, 316
- mandatory access control (MAC) models, 499
- mappings (implementation)
 - architecture-to-implementation consistency, 349–350, 377, 539
- components/connectors, 339
- configurations, 339
- design rationale, 339
- dynamic properties (behaviors), 339–340
- interfaces, 339
- non-functional properties, 340
- one-way/round-trip, 340
- Marcus Vitruvius Pollio. *See* Vitruvius
- matrix model (team structure), 669
- MCS (Grid Metadata Catalog Service), 175
- MDA. *See* Model Driven Architecture
- Ménage project, 619
- Message Tracing and Analysis Tool (MTAT), 237, 268, 287, 288, 544, 546
- message-oriented middleware, 37
- Metadata Catalog Service (MCS), 175
- MetaH architecture
 - description language, 225, 325. *See also* Architecture Analysis and Design Language
- MFC (Microsoft Foundation Classes), 36–37
- micromanagement, 662
- Microsoft Code Complete, 85
- Microsoft DCOM. *See* DCOM
- Microsoft Foundation Classes (MFC), 36–37
- Microsoft Internet Information Service (IIS), 496–497
- Microsoft PowerPoint. *See* PowerPoint
- Microsoft Robotics Studio (MRS), 442–443
- Microsoft Visio, 247, 255, 262, 645
- MIDAS wireless sensor network architecture, 444

- middleware
 - architecture
 - implementation
 - frameworks and, 343–346
 - CORBA. *See* CORBA
 - DCOM. *See* DCOM
 - message-oriented, 37
 - mismatches between styles and, 344–345
 - Prism-MW, 174
 - RMI. *See* RMI
 - RPC. *See* remote procedure calls
 - Siena publish-subscribe, 174
 - MIL. *See* module interconnection languages
 - Mindstorms (Lego), 443
 - MINLP (mixed-integer nonlinear programming), 399
 - MIP (mixed-integer linear programming), 399
 - mismatches (architectural)
 - connectors and, 158, 163, 164, 180–181
 - interfaces and, 72
 - between styles and middleware, 344–345
 - misrepresentation, 514, 519
 - mixed-integer linear programming (MIP), 399
 - mixed-integer nonlinear programming (MINLP), 399
 - MLS (multilevel security) environment, 499
 - mobile agents, 386, 406
 - mobile code styles, 116–118, 141, 155
 - Lunar Lander in, 118, 119
 - mobile computing, 405
 - mobility. *See also* deployment
 - architecture and, 405–408
 - challenges, 385–387
 - code, 386, 405–408
 - deployment and, 381–387, 408–409
 - model checking, 324
 - Model Driven Architecture (MDA), 348, 379, 650–651
 - model-based analysis, 322–327
 - modeling (architectural)
 - in architecture-driven deployment, 394–396
 - behavioral, 189
 - business case for, 245
 - complex, 188–189, 194–198, 242–243
 - components and, 188
 - concepts in, 188–189
 - concurrency, 190
 - configurations and, 188
 - connectors and, 188
 - definition, 76, 185
 - dynamic aspects of systems, 190–191
 - interfaces and, 188
 - of Internet-scale applications, 243
 - rationale and, 188, 191
 - stakeholder-driven, 186–187, 245
 - static aspects of systems, 190
 - of styles, 189–190
 - type systems and, 189
 - viewpoints and, 194–196
 - views and, 194–198
 - of Web, 188–189, 243
 - XML and, 233, 234
 - modeling notations (architectural). *See also* architecture
 - description languages as ADLs, 76
 - combinations of, 185, 194
 - definition, 76, 185
 - limitations of, 194
 - multiple, 229. *See also* extensible ADLs
 - range of, 185
 - selecting, 245
 - visualizations *vs.*, 288
 - modeling techniques. *See also* specific modeling techniques
 - ADLs, 209–242, 244. *See also* architecture description languages
 - domain-specific/style-specific, 244
 - evaluation rubric, 198. *See also* rubrics
 - extensible, 244
 - formal, 244
 - generic, 199–209. *See also* graphical modeling; natural language; Unified Modeling Language
 - Lunar Lander and, 199
 - summary table of, 244
 - syntactically rigorous, 244
 - models (architectural). *See also* specific models
 - accurate, 192–194
 - adaptation and, 539
 - ambiguous, 192, 193, 194
 - analysis of, 38–40, 291. *See also* analysis
 - definition, 76, 185
 - dynamic, 190, 260
 - dynamic aspects of system, 190–191
 - formal, 313
 - of functional aspects of systems, 191
 - inconsistencies in, 296–302
 - informal, 312
 - Linux (high level model), 300, 301, 302
 - of non-functional aspects of systems, 191
 - partial. *See* views
 - precise, 192–194
 - principal design decisions and, 200
 - semiformal, 312
 - static, 190
 - static aspects of system, 190
 - visualizations *vs.*, 250
 - waterfall, 25, 30, 42, 85

- model-view-controller (MVC) pattern, 95–97, 155
- modularity, 87
 - components and, 69
- module interconnection languages (MIL), 168
- modules, 58
- MonDe, 411
- morphological charts, 148
- MRS (Microsoft Robotics Studio), 442–443
- MTAT. *See* Message Tracing and Analysis Tool
- multilevel security (MLS) environment, 499
- multiple views, 196, 202, 206, 222, 251
- MVC pattern. *See* model-view-controller pattern
- The Mythical Man-Month* (Brooks), 669, 674
- Napster, 123, 426–427
- NASA
 - Apollo missions and, 99, 155
 - Cassini Web site, 528
 - Jet Propulsion Laboratory's Software Architect Program, 673
 - Peerdata project, 176
- natural language, 185, 199–201. *See also* modeling techniques
 - evaluation rubric, 200
 - Lunar Lander in, 200–201
 - programming language and, 199
 - restricted form of, 199
- network effects, 623
- new designs. *See* unprecedented design
- NFPs. *See* non-functional properties
- non-functional aspects of systems
 - design decisions and, 58
 - functional aspects of, 191
 - models of, 191, 198
- non-functional properties (NFPs), 312, 447–485
 - adaptation. *See* adaptation
 - availability, 478, 492
 - business case for, 483–484
 - complexity. *See* complexity
 - composite. *See* dependability
 - dependability
 - definition, 447
 - dependability, 478–483
 - efficiency. *See* efficiency
 - fault-tolerance, 478–479
 - implementation mapping for, 340
 - QoS and, 450
 - reliability, 447, 478
 - robustness, 478
 - safety, 479
 - scalability, 467–475
 - security. *See* security
 - survivability, 479
 - trust. *See* trust
- Norman, Don, 55
- Northrop, Linda, 606
- notational standards, 639–641
- notations. *See* modeling notations
- Nuseibeh, Bashar, 51
- OASIS (Organization for the Advancement of Structured Information Standards), 501
- Object Constraint Language (OCL), 204, 205, 206, 251, 255
- object diagram, 579–580, 581
- Object Management Group (OMG)
 - MDA, 348, 379, 650, 651
 - UML and, 640
- object request broker (ORB), 134, 135
- object skeleton, 133
- object stub, 133
- object-oriented design (OOD), 31–32
 - limitations of, 31–32
 - UML and, 31, 32
- object-oriented methods, 165
- object-oriented style, 103–104, 140. *See also* distributed objects style
 - Lunar Lander in, 104, 105
- object-oriented systems, UML viewpoints and, 204
- OCL. *See* Object Constraint Language
- OMG. *See* Object Management Group
- OmniGraffle, 201, 251, 255
- OMT, 203, 244, 279, 647, 648
- one-way mappings, 340
- OOD. *See* object-oriented design
- OOSE, 203
- open design, principle of, 494–495
- Open Group, 232
- Open Group Architecture Framework. *See* The Open Group Architecture Framework
- Open Source AADL Tool Environment (OSATE), 226
- operating system design, 106
- operational model, 583–584
 - control-flow diagrams, 585, 586
 - data-flow diagram, 584, 585
 - state-transition diagrams, 585, 587
- Operations activity layer, 541, 542
- ORB (object request broker), 134, 135
- Orca framework, 442
- Oreizy, Peyman, 539, 540
- Organization for the Advancement of Structured Information Standards (OASIS), 501

- OSATE (Open Source AADL Tool Environment), 226
- out-of-band knowledge, 515, 520
- P2P. *See* peer-to-peer
- P2P-based data distribution connectors, 176
- PAC. *See* presentation-abstraction-controller
- PACE (Practical Architectural Style for Composing Egocentric trust) style, 517–522
- threats and, 519–520
- PACE-based trust-enabled decentralized file-sharing application, 520–522
- packagers, DeLine's, 171
- Pappus, 26
- parallel approach (design strategy), 86
- partially automated analysis, 315
- partially ordered sets (POSETs), 213–214
- patches, 547
- A Pattern Language: Towns, Buildings, Construction* (Alexander, Ishikawa, and Silverstein 1977), 155
- patterns (architectural), 32, 73–76
- combination/modification/creation of, 138–139
- definition, 73, 94
- DSSAs v., 94
- examples, 94–99
- model-view-controller, 95–97, 155
- sense-compute-control, 97–99
- styles v., 73–74, 92–93, 137–139
- three-tier architecture (state-logic-display), 74–76, 94–95
- PDP (policy decision point), 501
- Peerdata project, 176
- peer-to-peer (P2P) styles, 123–124, 142, 426–432
- BitTorrent, 176, 432
- Gnutella, 123, 427–429, 512
- hybrid client-server/, 426–427
- Kazaa, 512
- Lunar Lander in, 124, 125
- Napster, 123, 426–427
- overlayed, 429–431
- P2P-based data distribution connectors, 176
- pure decentralized, 427–429
- resource trading, 432
- Skype, 429–431
- XREP, 511–514
- PEP (policy enforcement point), 501
- perfectionism, 661–662
- Perry, Dewayne, 100, 168
- persistent data access, 167
- persistent stores, 174
- perspectives (architectural), 65, 67
- cargo routing example, 65, 66
- definition, 65
- deployment, 65, 66, 67
- Peter, Laurence J., 666
- Peter Principle, 666, 674
- Petroski, Henry, 27, 55
- Philips Electronics, 16, 221, 608. *See also* Koala
- physical mobility, 405
- physical viewpoints, 195
- pipe-and-filter style, 14–15, 110–111, 140, 160, 222. *See also* Weaves
- architecture implementation frameworks for, 350–352
- batch-sequential v., 607
- filter programs and, 14
- Lunar Lander in, 111, 360–361
- simplicity of, 14
- pipes
- defined, 13
- UNIX, 169
- Player open-source project, 442
- plug-ins, 553, 554
- policy, 500
- policy decision point (PDP), 501
- policy enforcement point (PEP), 501
- Pollio, Marcus Vitruvius. *See* Vitruvius
- Polya, George, 26, 55
- Pompidou Center, 531
- Popkin System Architect. *See* Telelogic System Architect
- portability, definition, 468
- ports (Darwin), 210
- POSETS (partially ordered sets), 213–214
- PowerPoint (Microsoft). *See also* graphical modeling
- architectures, 154
- Koala architecture and, 274, 275
- Lunar Lander in, 202–203, 292
- modeling, 186, 201–203
- Practical Architectural Style for Composing Egocentric trust. *See* PACE style
- precision
- accuracy v., 192–193
- in evaluation rubric, 198
- prescriptive architecture, 59
- cargo routing example, 60–61, 62, 63
- presentation-abstraction-controller (PAC), 97
- Primary Functional Need, 148

- principal*, 58
- principal design decisions
 - architecture and, 1, 20, 58, 185, 189, 191, 192, 209
 - models and, 200
- principals, 499
- Prism-MW middleware, 174
- privileges, 500
- problem domains. *See* domains
- problem spaces, 567, 570, 571, 572
- procedure call connectors, 71, 165–166
- procedure calls, 71. *See also* remote procedure calls
- process standards, 648–651
- processes. *See* design processes; development processes
- processing, 68
- processing elements, 67
- product lines (product families), 566, 591–608
 - architecture and, 15–19, 591–592
 - architectures, 594–600
 - business, 592–593
 - defined, 15
 - DSSAs, styles, and, 606–608
 - engineering, 593–594
 - hybrid, 606
 - implementing, 602–604
 - Koala and, 16–19
 - of Lunar Lander, 601
 - organizational issues, 606
 - Philips TVs as, 17
 - recovery and, 605
 - reference architecture and, 58
 - reuse and, 18, 605
 - SDRs and, 614
 - selection and, 599
 - Software Product Lines: Practices and Patterns*, 606
 - source-code management systems and, 603–604
 - unifying, with different intellectual heritage, 604–606
 - what-if analysis and, 600–601
- product populations, 18, 19, 566
- product-line motivations, for adaptation, 527–528
- profiles (UML), 204, 641
- program families. *See* product lines
- programming languages. *See* also modeling techniques; *specific programming languages*
- imperative, 102
- natural language and, 199n
- psychological acceptability, principle of, 495–496
- public key cryptography, 490
- publish-subscribe middleware, Siena, 174
- publish-subscribe styles, 118–120, 141
 - Lunar Lander in, 120, 121
- QoS (quality of service), 389, 450
- QoSA (Quality of Software Architectures), 81
- qualifications, of architect (Vitruvius), 3–4
- quality of service (QoS), 389, 450
- Quality of Software Architectures (QoSA), 81
- query mechanisms, 167
- quiescence, 559
- radios, software-defined. *See* software-defined radios
- Rapide, 185, 213–219
 - effect visualizations and, 268
 - evaluation rubric for, 214–215
 - Lunar Lander in, 215–218, 281, 293
 - POSETs, 213–214
 - Simulator, 325
 - visualizations, 268, 280–281
- Rational Rose, 276, 279, 645, 646
- Rational Unified Process (RUP), 648–650
- rationale, 67, 188
 - modeling and, 188, 191
- RBAC (role-based access control) model, 498
- RCS, 601
- Rechtin, Eberhardt, 187, 246
- recovery
 - (architectural/design), 77, 79, 139, 142–143
 - Apache Architecture Recovery Project and, 143
 - clustering and, 142–143
 - definition, 77
 - Linux and, 143
 - product lines and, 605
- redeployment, 381, 411
- reference architectures, 93, 572
 - choosing form of, 590–591
 - classes of, 589
 - definition, 58, 588
 - developing, 589–591
 - DSSAs and, 588–591
 - for Lunar Lander, 591
 - product lines and, 58
- Reference Model of Open Distributed Processing. *See* RM-ODP
- reference monitor, 498
- reference requirements, 29
 - DSSA, 585–588
 - Lunar Lander and, 588
- refined experience, 87, 90–91. *See also* patterns; styles
- in action, 91–143
 - conception without, 144–149
 - recovery and. *See* recovery
 - unprecedented design v., 150–151
- refinement inconsistencies, 196
- in models, 300, 302

- The Reflective Practitioner* (Schön), 55, 155
- release, 400–401
- Release Dock, 411
- releases, 547
- reliability, 478
- Remos, 411
- remote evaluation, 406
- remote procedure calls (RPC), 108, 135, 158, 165, 171
- removing mental blocks (transformation strategies), 148–149
- replication, REST and, 419–420
- repositories, software component, 167
- representation model, 583, 584
- REpresentational State Transfer. *See* REST
- repudiation attacks, 490, 515
- reputation, 510. *See also* trust
- reputation-based systems, 510–514
- requirements
- isolation of, 26–27
 - venture capitalists and, 27–28
- requirements engineering, 25–29, 52
- design and, 25–29, 150–151
 - software architectures v., 150
- requires rule, 180
- resource trading P2P, 432
- resources, 388, 499–500
- exclusive, 388
 - sharable, 388
- REST (REpresentational State Transfer), 12, 138, 175, 416–422, 443–444
- components and, 70
 - data elements and, 67
 - derivation of, 417–418
 - dynamic extension in, 420
 - layered separation, 418–419
 - limited commonality and, 420
 - replication and, 419–420
 - Web v., 420n
- restricts rule, 180–181
- reuse
- components and, 69
 - DSSE and, 616–617
 - Koala and, 17, 18, 19
 - product lines and, 18, 605
 - software architecture and, 18
 - software, design and, 152
- reuse-oriented robotic architectures, 442
- reverse engineering, 494
- reviews, 317–322
- RMI (Java), 175, 344, 378, 379
- connectors and, 157
 - distributed objects and, 133
- RM-ODP (Reference Model of Open Distributed Processing), 637–639
- robotics, 437–443
- architectures, 438–443
 - development frameworks for, 442–443
 - Lego Mindstorms and, 443
 - MRS and, 442–443
 - Orca and, 442
 - Player open-source project, 442
- robustness, 478
- role-based access control (RBAC) model, 498
- Rose. *See* Rational Rose
- Rosenblum, David, 166
- round-trip mappings, 340
- routing, 172
- RPC. *See* remote procedure calls
- rubrics (evaluation)
- for AADL, 226
 - for Acme, 230
 - for Darwin, 210
 - for graphical modeling, 201–202
 - for graphical visualizations, 275–276
 - for Koala, 222
 - for LTSA, 284
 - for modeling techniques, 198
- for natural language, 200
 - for Rapide, 214–215
 - for Rapide visualizations, 280–281
 - for textual visualizations, 273–274
 - for UML, 205–206
 - for UML visualizations, 279–280
 - for visualization techniques, 272
 - for Weaves, 223
 - for Wright, 219
 - for xADL, 237
 - for xADL visualizations, 287–288
- rule-based/expert systems
- designing and, 91
 - style, 112–115, 141
- RUP. *See* Rational Unified Process
- SADL, 325
- safety, 479
- SCA. *See* Software Communications Architecture
- SCA Reference Implementation (SCARI), 615
- scalability, 467–475
- components and, 468–471
 - configurations and, 473–475
 - connectors and, 471–473
 - definition, 467–468
- SCARI (SCA Reference Implementation), 615
- SCC pattern. *See* sense-compute-control pattern
- scenario-based analysis, 314
- schemas, xADL, 234–235
- Schneider, Bruce, 496
- Schön, Donald, 55, 155
- scripting languages, 555
- SDRs. *See* software-defined radios

- Secrets and Lies* (Schneier), 496
- secure cooperation, architectural access control and, 505
- Secure xADL, 501–503
- security, 487–508
- access control models, 498–499
 - architectural access control, 497–522
 - aspects of, 489–492
 - availability and, 492
 - business case for, 522–523
 - confidentiality and, 489
 - contract, 500, 501
 - cryptography and, 490–491
 - definition, 489
 - design principles for, 492–497
 - encryption and, 488, 490
 - for IIS, 496–497
 - integrity and, 491–492
 - MLS environment, 499
 - as process, 496
 - Schneier on, 496
 - threats. *See* threats
 - trust and. *See* trust
- semantic clustering, 143
- semantic interconnection, 169
- semiformal models, 312
- sense-compute-control (SCC) pattern, 97–99
- Lunar Lander and, 98–99
- sense-plan-act (SPA) robotic architectures, 438–439
- sensor-controller-actuator. *See* sense-compute-control pattern
- separation of concerns, 90
- product lines and, 603
- separation of privilege, principle of, 495
- sequence diagrams, 207, 208
- service classes
- communication, 163
 - conversion, 164
 - coordination, 163–164
 - facilitation, 164
- service packs, 547
- service-oriented architectures (SOAs), 433–436
- sharable resources, 388
- shared data access, 71
- shared key cryptography, 490
- shared memory stores, 174
- shared memory styles. *See* shared state styles
- shared state styles, 111–115, 141
- blackboard, 112, 141
 - rule-based/expert systems, 91, 112–115, 141
- shearing layers, 529–532
- Siena publish-subscribe middleware, 174
- signature-based authentication, 515
- signatures, digital, 490, 519
- silver bullets, 91
- simple connectors, 164, 165
- simple machines, 88. *See also* abstraction
- Simple Object Access Protocol (SOAP), 174, 175, 436
- simulation, design by, 151
- simulation-based analysis, 328–333
- Skype, 429–431
- SOAP. *See* Simple Object Access Protocol
- SOAs. *See* service-oriented architectures
- software
- architects. *See* architects
 - architecture. *See* architectures
 - components. *See* components
 - connectors. *See* connectors
 - developers. *See* developers
 - development process. *See* development processes
 - evolution. *See* evolution/maintenance
 - maintenance. *See* evolution/maintenance
 - reuse. *See* reuse
 - systems. *See* systems
- Software Communications Architecture (SCA), 612–614
- software component repositories, 167
- software designers, architects as, 658
- software engineering. *See also* design; development processes; domain-specific software engineering; implementation architecture and, 23–55
- design phase, 29–30. *See also* design
 - requirements activity, 25–29, 52, 150–151
- software engineering economists, architects as, 661
- Software Engineering Institute, 81
- software interconnection models, 168, 184
- Software Product Lines: Practices and Patterns* (Clements and Northrop), 606
- software technologists, architects as, 659–660
- software-defined radios (SDRs)
- CORBA and, 151
 - product lines and, 614
 - SCA and, 612–614
 - SCARI and, 615
- solution spaces, 570, 571, 572
- source-code management systems, product lines and, 603–604
- SPA robotic architectures. *See* sense-plan-act robotic architectures
- spaghetti-code, 377
- spiral model, 42, 53, 85, 648
- SPKI (Standard Public Key Infrastructure), 522
- SQL, 167

- stack/heap memory access, 167
- stakeholder-driven modeling, 186–187
- strategy, 245
- stakeholders (architectural)
 - analysis and, 315–317
 - architects as, 79, 316
 - architects' relations with, 664, 669–672
 - customers as, 79, 316–317
 - developers as, 79, 316
 - managers as, 79, 316
 - vendors as, 317
- standard I/O framework (studio), 341, 350–351
- Standard Public Key Infrastructure (SPKI), 522
- standards, 621–654
 - business case for, 652–653
 - conceptual, 194, 253, 627–639
 - de facto, 622, 625
 - de jure, 622, 625
 - definition, 622
 - drawbacks of, 624–625
 - network effects and, 623
 - notational, 639–641
 - process, 348, 379, 648–651
 - reasons for usage, 623–624
 - sources for, 622
 - standard UML tools, 645–647
 - SysML, 410, 641–644
 - Telelogic System Architect, 647–648
 - UML as, 640–641
- standards compliance experts, 660–661
- state, 68, 543
 - definition, 543
 - observing/collecting, 543–545
- statecharts, 190, 203
 - UML, 207
- stateful mobility, 405
- stateless mobility, 405
- state-logic-display. *See* three-tier architecture
- state-transition diagrams, 585, 587
- static analysis, 313
- static models, 190
- studio. *See* standard I/O framework
- stepwise refinement, 89, 154
- stereotypes (UML), 204, 205, 280, 640, 641, 646
- stream connectors, 111, 169–170
- streams, 169
- Strom, Robert, 171
- strong mobility, 405
- structural characteristics, of
 - software architecture, 311
- structural views, 67
- styles (architectural), 4–5, 72–73
 - adaptation and. *See* adaptation
 - applied. *See* applied architectures/styles
 - Big Ball of Mud, 137
 - C2, 124, 126–131, 138, 142, 155, 352–359
 - categorization of, 101
 - combination/modification/creation of, 138–139
 - complex, 124, 126, 142
 - constraints and, 5, 138
 - data-flow, 14–15, 109–111, 140, 160, 222, 350–352, 607
 - definition, 73, 99, 189
 - design decisions and, 73, 99, 189
 - distributed objects, 124, 126, 142
 - implicit invocation, 118–123, 141
 - interpreter, 115–118, 119, 141, 155
 - language-influenced, 102–104, 140
 - layered, 104–109, 140, 418–419
 - Lunar Lander and, 101–102
 - mismatches between
 - middleware and, 344–345
 - modeling of, 189–190, 198
 - PACE, 517–522, 524
 - patterns and, 73–74, 92–93, 137–139
 - peer-to-peer, 123–124, 125, 142, 176, 426–432
 - Perry and Wolf on, 100
 - product lines, DSSAs, and, 606–608
 - as 'refined experience in action,' 137
 - shared state, 91, 111–115, 141
 - simple, 101
 - summary chart, 140–142
 - Vitruvius and, 5, 91
 - of Web, 11–12
 - YAAS, indeed, 139
- style-specific ADLs, 221–228
- style-specific/domain-specific
 - modeling techniques, 244
- subdomains, 566, 567
- subjects, 499
- subsumption robotic
 - architectures, 439–440
- subsystem-level analysis, 305–307
- Subversion, 601
- survivability, 479
- Swing framework (Java), 36–37, 69, 96
- switching, 172
- syntactic clustering, 143
- syntactic interconnection, 169
- SysML, 410, 641–644
 - parametric diagram, for
 - Lunar Lander, 643
 - requirements diagram, for
 - Lunar Lander, 642
 - UML and, 642–643
- System Architect (Telelogic), 647–648
- system-level analysis, 305–307
- Szyperski, Clemens, 70

- tagged values, 204, 205, 280, 640, 641, 646
- taxonomy, for connectors. *See* classification
- TCP/UDP communication sockets, 169
- teams, 665–669. *See also* architects
- teams (architects), 665–669
 - allegiance to the
 - organization, 667–668
 - allegiance to the project, 666–667
 - balance of skills, 665–666
 - duration of the
 - involvement, 668
 - structure, 669
- Telelogic System Architect, 647–648
- televisions (TVs)
 - domains, airplanes and, 565, 566, 567
 - as product lines, 17
 - software architecture for, 18
- temporal logic notations, 313
- terminology (software architecture), 58–76. *See also* specific definitions
 - business case for, 80
 - informal terms v., 57
 - precise usage, 80
- testing, analysis and, 38–40. *See also* analysis
- textual visualizations, 251–252
 - evaluation rubric for, 273–274
- The Open Group Architecture Framework (TOGAF), 634–637
- threats
 - addition of unknowns, 514–515, 519–520
 - collusion, 514, 519
 - denial of service, 514, 519
 - fraudulent actions, 514, 519
 - impersonation, 514, 519
 - misrepresentation, 514, 519
 - PACE and, 519–520
- three-layer robotic architectures, 440–442
- three-tier architecture (state-logic-display), 74–76, 94–95
 - examples, 75, 76
 - graphical view, 74
- The Timeless Way of Building* (Alexander), 20, 22, 154–155
- timing, adaptation and, 536–537
- Tinker Toys, 14
- TOGAF. *See* The Open Group Architecture Framework
- top-down design, 148, 154
- traceability, 338, 339
- Tracz, Will, 619
- transfer of control, 157, 163, 182
- transfer of data, 163, 165, 185
- transformation step, 144–145
- transformation strategies (removing mental blocks), 148–149
- transient data access, 167
- Trojan horses, 512
- true object, 134
- trust, 508–522
 - comparable, 515
 - context and, 509
 - deciding who to trust, 515, 520
 - definitions, 509
 - expression of, 516
 - making trust visible, 516
 - reputation and, 510
 - reputation-based systems and, 510–514
 - threats. *See* threats
 - values for, 510
- trust boundary (Firefox), 506–507
- trust management, 508–509
 - business case for, 523
 - decentralized, architectural approach to, 514–522
- trust models
 - definition, 510
- eBay, 511
- XREP, 511–514
- Tufte, Edward, 263–264
- turbine visualization, 43–48
 - of agile development process, 49–51
 - DSSA-based project, 49
- Twin Peaks model of development, 51, 55
- type systems, modeling and, 189
- UDP technologies, 169, 175
- UML. *See* Unified Modeling Language
- 'underline the nouns'
 - underline the verbs' design approach, 151
- UniCon, 325
- Unified Modeling Language (UML), 203–209
 - as ADL, 209
 - composite structure diagram, 255
 - constraints for, 204
 - designing and, 91
 - diagram types, 257
 - diagrams (viewpoints), 203, 204, 205, 640–641
 - evaluation rubric for, 205–206
 - Lunar Lander in, 206–209
 - OMG and, 640
 - OOD and, 31, 32
 - profiles, 204, 641
 - sequence diagrams, 207, 208
 - as standard, 640–641
 - standard, tools, 645–647
 - statechart, 207
 - stereotypes, 204, 205, 280, 640, 641, 646
 - strengths of, 203
 - SysML and, 642–643
 - tagged values, 204, 205, 280, 640, 641, 646
 - visualizations, 276–280
- Unified Process, 54, 85
- unit interconnection, 168

- UNIX
 - command-line shell
 - programs, 13
 - filter programs, 13, 14
 - pipes, 169
 - X Windows, 167
 - unprecedented design,
 - 144–149
 - analogy searching, 146–147
 - brainstorming, 147
 - convergence step, 144–145
 - divergence step, 144–145
 - experience v., 150–151
 - Jones on, 144, 145, 146, 148
 - literature searching,
 - 147–148
 - morphological charts, 148
 - removing mental blocks
 - (transformation strategies), 148–149
 - strategies, 144–149
 - transformation step,
 - 144–145
- use-case diagrams, 583
- vendors, as stakeholders,
 - concerns of, 317
- Venn diagram, product-line
 - architecture as, 594
- venture capitalists,
 - requirements and,
 - 27–28
- Verhoef, Chris, 68
- version control systems, 601
- versions, 388
- view coordination, 260–261
 - visualizations and, 260–261
 - visualizations rubric and,
 - 272
- viewpoints
 - behavioral, 196
 - concurrency, 196
 - definition, 195
 - deployment, 195
 - in evaluation rubric, 198
 - logical, 195
 - modeling and, 194–196
 - physical, 195
 - UML, 203, 204, 205
 - views v., 195
- views, 194–196
 - consistency among,
 - 196–198
 - definition, 194
 - deployment, 195
 - in evaluation rubric, 198
 - inconsistencies among,
 - 196–198
 - of Lunar Lander, 195, 197
 - modeling and, 194–198
 - multiple, 196, 202, 206,
 - 222, 251
 - structural, 67
 - viewpoints v., 195
 - visualizations v., 255–259
- virtual channel, 181. *See also*
 - ducts
- virtual machines style,
 - 105–108, 140
 - Lunar Lander in, 107–108
- virtual memory translation,
 - 171
- viruses, 429, 512
- Visio (Microsoft), 247, 255,
 - 262, 645
- visualizations, 249–290
 - aesthetics of, 261
 - business case for, 289
 - canonical, 250–251
 - comprehensibility in, 260
 - consistency in, 259–260
 - constructing, 261–263
 - coordinating, 264–266
 - dynamic usage of, 266, 268
 - dynamism in, 260
 - effect, 266, 268
 - evaluating, 259–261
 - extensibility, 261
 - fidelity in, 259
 - graphical, 253–255,
 - 274–276
 - hybrid, 255, 256
 - L TSA, 282–284
 - modeling notations v., 288
 - models v., 250
 - Rapide, 268, 280–281
 - rubric for. *See* rubrics
 - techniques, 272–278. *See also* rubrics
 - textual, 251–252, 272–274
 - Tufte and, 263–264
 - turbine, 43–48
 - UML, 276–280
 - view coordination in,
 - 260–261
 - views v., 255–259
 - xADL 2.0, 284–288
- Vitruvius, 3–4, 5
 - De Architectura*, 3
 - on architecture, 3–5, 20, 91
- W3C. *See* World Wide Web Consortium
- washing machines,
 - development of, 26,
 - 28, 55
- waterfall model, 25, 30, 42, 85
- weak mobility, 405
- Weaves, 221, 222–225
 - evaluation rubric for, 223
 - Lunar Lander in, 224–225
 - as pipe-and-filter style, 222
- Web
 - architectural style of, 11–12
 - architecture of, 7–12, 22
 - Berners-Lee and, 22
 - as distributed hypermedia
 - application, 417, 424
 - HTTP protocol and, 8,
 - 9–10, 95, 243
 - IETF and, 22
 - modeling of, 188–189, 243
 - MVC paradigm and, 96–97
 - REST v., 420n
 - W3C and, 22
- Web services
 - acronyms/jargon, 436
 - specifications, 436
- Web services Death Star
 - (WS-*), 436
- Web-based applications, 95
- WebDAV requests, 496, 497
- what-if analysis, product lines
 - and, 600–601
- WICSA (Working IEEE/IFIP Conference on Software Architecture), 81
- wireless sensor networks
 - (WSNs), 383, 389,
 - 443, 444

- Wolf, Alexander, 100, 166
- Working IEEE/IFIP
 - Conference on Software Architecture (WICSA), 81
- World Wide Web. *See* Web
- World Wide Web Consortium (W3C), 22, 627
- Wright, 219–221
 - CSP formalism, 219, 220, 313
 - evaluation rubric for, 219
 - Lunar Lander in, 219–220
 - pipe connector modeled in, 305
- WS-* (Web services Death Star), 436
- WSNs (wireless sensor networks), 383, 389, 443, 444
- X Windows, 167
- XACML (eXtensible Access Control Markup Language), 501, 502
- xADL 2.0, 234–242. *See also*
 - Secure xADL
 - data binding library, 235
 - data binding library generator (Apigen), 235
 - evaluation rubric for, 237
 - Lunar Lander in, 237–242
 - schemas, 234–235
 - visualizations, 284–288
- xADLite, 235–236, 284
 - Lunar Lander in, 258–259, 312
- XMI. *See* XML metadata interchange
- XML (Extensible Markup Language), 226
 - document type definition, 233
 - modeling and, 233, 234. *See also* Architecture Description Markup Language; xADL
- XML metadata interchange (XMI), 171, 277, 278, 279
- XPCOM, 505
- XPConnect, 506
- XREP, 511–514
- XTEAM. *See* eXtensible Tool-chain for Evaluation of Architectural Models
- Y2K problem, 563
- YAAS, (yet-another-architectural-style), 139
- Yellin, Daniel, 171
- yet-another-architectural-style (YAAS), 139
- zipper, development of, 27

This Wiley Student Edition is part of a continuing program of paperbound textbooks especially designed for students in developing countries at a reduced price.

THIS BOOK IS FOR SALE ONLY IN THE COUNTRY TO WHICH IT IS FIRST CONSIGNED BY WILEY INDIA PVT. LTD. AND SHOULD NOT BE RE-EXPORTED.

**FOR SALE ONLY IN:
INDIA, BANGLADESH,
NEPAL, PAKISTAN, SRI
LANKA & BHUTAN**

WILEY

Wiley India Pvt. Ltd.

4435/7, Ansari Road, Daryaganj
New Delhi-110 002
Customer Care +91 11 43630000
Fax +91 11 23275895
csupport@wiley.com
www.wileyindia.com
www.wiley.com



X0048ASA1P

Software Architecture: Foundations, Theory,...
Used, Good